EUSEC
EU SECURITY CERTIFICATION

EUROPEAN SECURITY CERTIFICATION FRAMEWORK

D3.4 ARCHITECTURE AND TOOLS – INTEGRATION FRAMEWORK

V 1.0

PROJECT NUMBER: 731845

PROJECT TITLE: EU-SEC

DUE DATE: 30/06/2018	DELIVERY DATE: 30/06/2018
AUTHOR: Philipp Stephanow-Gierach	PARTNERS CONTRIBUTED: Fabasoft, MFSR, CSA, SixSq
DISSEMINATION LEVEL:* PU	NATURE OF THE DELIVERABLE:** R
INTERNAL REVIEWERS: SI-MPA, NIXU	

*PU = Public, CO = Confidential

**R = Report, P = Prototype, D = Demonstrator, O = Other

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731845

EXECUTIVE SUMMARY

This deliverable describes an integration framework for the tools needed to implement continuous security audits supporting cloud service certification. This framework is based on three pillars: The first one consists of describing the interaction of existing techniques which are available as background in the EU-SEC project. The result is a tool chain where each component is based on the specifications described in Deliverable 3.1, 3.2 and 3.3.

The second pillar of the integration framework consists of a risk-driven process describing how to integrate the tool chain with existing cloud services. The steps of this risk-driven integration process include selecting a global integration strategy, discovering cloud service, deriving feasible measurement techniques, selecting of suitable metrics, deploying components of the tool chain, and adapting measurement techniques to changes of the cloud service under audit at runtime. Example application of the integration process to produce evidence and measurement results on application level as well as on platform level are described.

The third pillar of the integration framework aims at quantifying inaccuracy in measurement results produced by continuous test-based measurement techniques because erroneous results undermine the trust placed in objective evaluation and resulting claims. To that end, a method is presented which permits to evaluate accuracy and precision of measurement results which allows comparing alternative techniques as well as alternative technique's configurations. An example application of this approach is demonstrated where a cloud provider is given a set of candidate configurations for a particular test-based technique and selects the most suited one.

DISCLAIMER

ACRONYMS

AWS	Amazon Web Services
CAIQ	Consensus Assessments Initiative Questionnaire
CCM	Cloud Control Matrix
CMIS	Content Management Interoperability Services
CRIME	Compression Ratio Info-leak Made Easy
CSA	Cloud Security Alliance
CSP	Cloud Service Provider
CTP	Cloud Trust Protocol
DDoS	Distributed Denial of Service
DSL	Domain-specific language
EBS	Amazon Elastic Block Storage
EC2	Amazon Elastic Compute Cloud
EU-SEC	European Security Certification Framework
IaaS	Infrastructure-as-a-Service
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
KMS	AWS Key Management Service
RDS	Amazon Rational Database Service
S3	Amazon Simple Storage Service
SLO	Service Level Objective
SOAP	Simple Object Access Protocol
SQO	Service Qualitative Objective

SSL	Secure Sockets Layer
TLS	Transport Layer Security
URL	Uniform Resource Locator
VDE	Virtual Development Environment
VM	Virtual Machine
WebDAV	Web-based Distributed Authoring and Versioning
XML	Extensible Markup Language

Table of contents

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Integration of continuous security audits with existing cloud services to support continuous certification requires to consider the following key aspects:

INTEGRATION OF TOOLS NEEDED TO ENABLE CONTINUOUS SECURITY AUDITS

Multiple applications are required to implement continuous security audits of cloud services. As already pointed out in Deliverable 3.1, 3.2 and 3.3, these applications include: Objective evaluation application (Deliverable 3.1), continuous measurement techniques (Deliverable 3.2) as well as evidence stores (Deliverable 3.3). These applications have to interact in a well-defined manner to enable continuous security audits, that is, they have to be integrated with each other to implement the *tool chain* required for continuous cloud security audits.

INTEGRATION OF THE TOOL CHAIN WITH EXISTING CLOUD SERVICES

In order for the tool chain to become meaningful, it has to interact with existing cloud services in a well-defined way such that evidence and measurement results are produced (and stored) supporting the validation of controls. Integrating the tool chain with a cloud service is not confined to integrating the continuous measurement technique used to produce evidence and compute measurement results. It also has to address questions such as: Where to host the evidence store? How to handle changes of the configuration or composition of cloud service under audit? Where to host the claim store?

EVALUATION OF ACCURACY AND PRECISION OF MEASUREMENT RESULTS

The tool chain continuously, i.e., automatically and repeatedly produces and stores measurement results to support validation of controls of security certificates. Inaccurate results undermine both cloud provider's and customer's trust: On the one hand, measurement results that incorrectly indicate satisfaction of a control erode customer's trust. On the other hand, cloud service providers may dispute results incorrectly suggesting that controls are not fulfilled. Therefore, it is essential to evaluate the accuracy and precision of measurement results produced by continuous test-based measurement techniques, that is, how close are produced results to their true values?

Consider, as an example, the following extract of control *TVM-02: Threat and Vulnerability Management Vulnerability / Patch Management* of CSA's CCM (1):

"Policies and procedures shall be established, and supporting processes and technical measures implemented, for timely detection of vulnerabilities within organizationally-owned or managed applications, infrastructure network and system components (e.g., network vulnerability assessment, penetration testing) […].".

One possibility to produce measurement results supporting validation of this control consists of a test-based technique which executes a vulnerability scanner every ten minutes and checks whether no vulnerability is found. The question is now whether this technique makes mistakes by, e.g., incorrectly suggests that the cloud service under test has no vulnerabilities while it actually has. In this case, it unclear to what extent the produced results can be used to check the control. Does, e.g., the vulnerability scanner only occasionally miss detecting a particular vulnerability or does it never detect it?

1.1 SCOPE AND OBJECTIVE

This deliverable's main objective is to describe an integration framework for the tool chain which is needed to implement continuous security audits supporting cloud service certification.

As already outlined above, integrating this tool chain first of all requires to integrate existing techniques following the specifications described in Deliverable 3.1, 3.2 and 3.3 with each other. To that end, a subgoal of this deliverable consists of delineating the different components of the tool chain as well as describing their interaction.

Furthermore, the tool chain to implement continuous cloud security audits has to be integrated with existing cloud services. Therefore, another subgoal of this deliverable is to describe a risk-driven integration process which considers different levels of integration, derivation of feasible measurement techniques, selection of and suitable metrics, deployment strategies of the tool chain as well as adaption of measurement techniques at runtime.

Finally, measurement results produced by measurement techniques contain the essential information to determine of a cloud service satisfies a set of SLOs or SQOs. Inaccurate measurement results therefore undermine the trust placed in objective evaluation and resulting claims. Thus, the last subgoal of this deliverable is to provide a method to evaluate the accuracy and precision of measurement results produced by continuous test-based measurement techniques. This method permits to compare alternative techniques as well as alternative technique's configurations.

1.2 WORKING PACKAGE DEPENDENCIES

The integration framework introduced in this document has dependencies with Task 3.1, 3.2, 3.3 as well as with Task 5.1 of Working Package 5 (see Figure 1-1). Consider Task 3.1 which specifies data structures and protocols used to store and evaluate instances of control objectives. One example implementation of this specification is the CTP API which has been developed by CSA. This specification forms the basis for one component of the tool chain described in Section 2 of this deliverable. Furthermore, the data structures defined for objective evaluation in Deliverable 3.1 can serve as the starting point to conduct a risk analysis whose results are required to decide where to host the objective evaluation application during Step 5 *Deployment of the tool chain* of the risk-driven integration process (see Section 3.2). Similarly, Deliverable 3.3 serves as input to the risk-driven integration process: It define a common data structure to represent evidence, i.e., instances of evidence produced by (test-based) measurement techniques. This data structure depicts the starting point to investigate what additional risk exposure is incurred through storing evidence and this guides the decision where to persist evidence, i.e., where to deploy the evidence store.

Moreover, consider Task 3.2 which develops a domain specific language (DSL) called *ConTest* which allows rigorously defining continuous test-based measurements. This unified configuration language is crucial when comparing accuracy and precision of alternative test-based techniques as well as alternative configurations because ConTest standardizes configuration representation. This means that ConTest provides a standardized way how to refer to a specific (configuration of a) continuous test-based measurement technique which is necessary for explicit, unambiguous comparison of alternative techniques and alternative techniques' configuration.

Furthermore, the risk-driven integration process of the integration framework (see Section 3.2) presented in this deliverable serves as input to Task 5.1 of Working Package 5 which centers around the preparation of the pilot implementing continuous security audits. Once the pilot has been prepared considering the risk-driven integration process, the process description will be revised according to necessary alterations observed during the pilot.

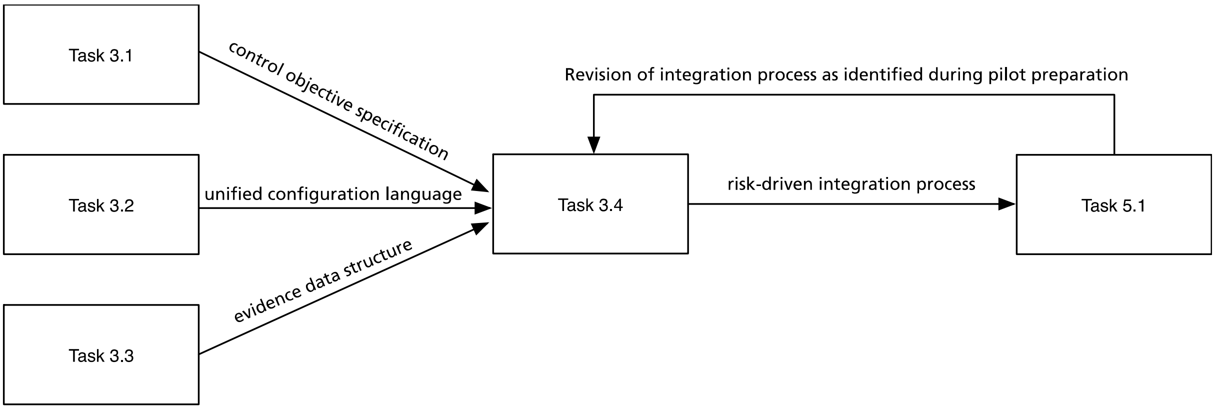

Task 3.1
Task 3.2
Task 3.3
control objective specification
unified configuration language
evidence data structure
Task 3.4
Revision of integration process as identified during pilot preparation
risk-driven integration process
Task 5.1

Figure 1-1: Dependencies of Task 3.4

1.3 ORGANISATION OF THE DELIVERABLE

The remainder of this document is organized as follows: The next section outlines how existing tooling within the EU-SEC project interacts in order to implement continuous cloud security audits supporting cloud certification. Thereafter, Section 3 describes the steps of integrating the tool chain with existing cloud services. Following this integration process, Section 4 and Section 5 describe example integrations to produce evidence on the application level and platform level, respectively. These examples are driven by the pilot requirements elicited as part of Task 5.1 of Working Package 5. Section 6 then presents an approach to experimentally evaluate the accuracy and precision of continuous test-based measurement techniques. Finally, Section 7 concludes this deliverable.

2 TOOL CHAIN

This chapter outlines existent tooling and solutions within the EU-SEC project and describes how they interact with each other in order to allow for continuous, i.e., automated and repeated security audits. Sections 2.1 - 2.4 outline tools involved in the tool chain while Section 2.5 describes how these tools interact, thereby composing the tool chain required for continuous cloud security audits.

2.1 CLOUDITOR

The Clouditor toolbox consists of five main components which are shown in Figure 2-1. It can be used to design and execute continuous test-based assurance techniques. The test results serve as input to compute test metrics which, in turn, can be used as evidence to support validation of controls.

The Engine and the Explorer are responsible for continuously executing and adapting assurance techniques. The Simulator and the Evaluator are used prior to deployment, they serve to select techniques and respective configurations which are most suitable to check if a cloud service complies with a particular requirement set. Lastly, the components can be viewed and configured from a Dashboard. Each component is designed as a micro-service and can be deployed in an individual container.

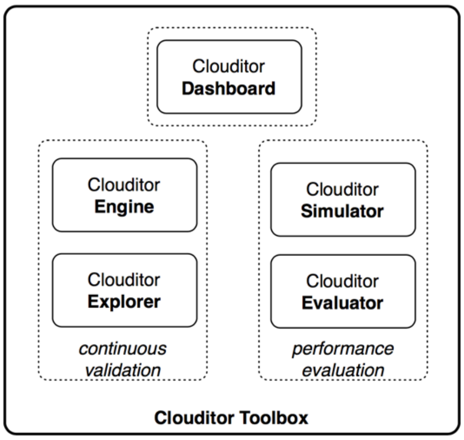


Figure 2-1 Tools of the Clouditor Ecosystem

In the following, we will only outline components of continuous validation, i.e., the Clouditor Engine and the Clouditor Explorer.[1] The Clouditor Engine implements and deploys test-based assurance techniques. It consists of test suites which comprise test cases, workflows which model dependencies between test suite executions, and metrics which are used to reason about the sequence of results of test suite executions. Figure 2-2 shows a high-level architecture of the Clouditor Engine's components, including data and control flow.

Discovering a cloud-based application's interfaces and configuring the selected assurance technique is the task of the *Clouditor Explorer*. To that end, the Explorer discovers cloud services' composition and interfaces at runtime as well as automatically generates and adapts test configurations.

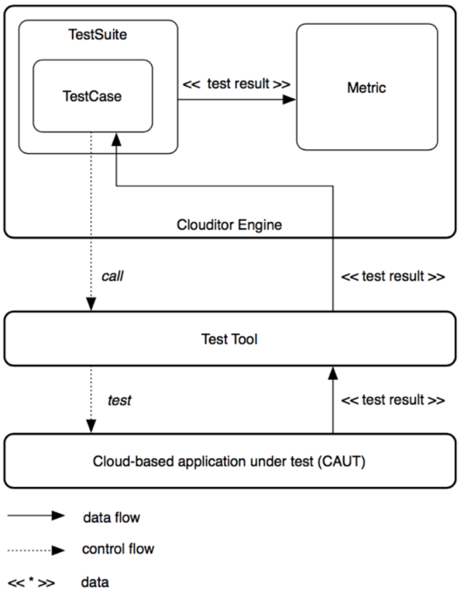


Figure 2-2 Overview of Clouditor's Engine main components (with external test tool)

2.2 THE CTP SERVER

To avoid any misunderstanding, it is important to highlight the difference between the "CTP API" and "the CTP Server". While the "CTP Data Model and API" (or "CTP API" for short) defines a protocol specification[2] and data model, the "CTP Server" defines a tool that implements the

---

[1] For further details on the remaining components of the Clouditor Toolbox see https://www.aisec.fraunhofer.de/content/dam/aisec/Dokumente/Publikationen/Studien_TechReports/englisch/Whitepaper_Clouditor_Feb2017.pdf.

[2] http://htmlpreview.github.io/?https://github.com/cloudsecurityalliance/ctpd/blob/master/client/CTP-Data-Model-And-API.html

"CTP API" itself. The CTP Server was initially created as a proof of concept to validate the "CTP Data Model and API".

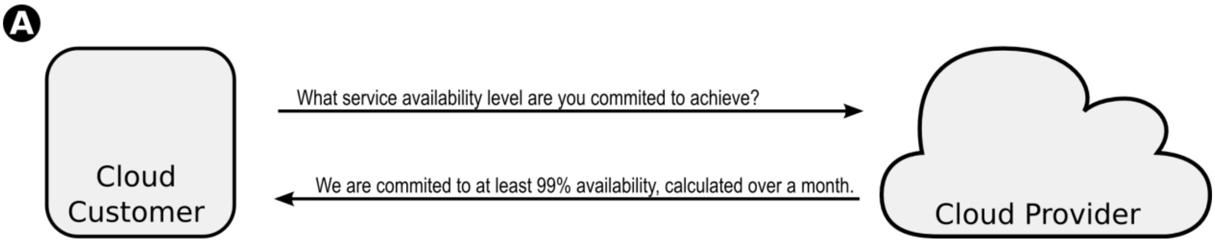
In the following, we will first present an overview of the CTP API and then describe how it was implemented as a tool through the CTP Server.

2.2.1 THE CTP API PRINCIPLES

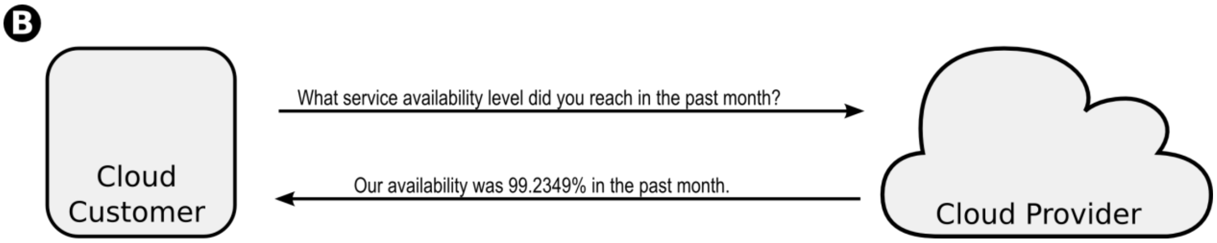
The CTP API is designed to be a RESTful protocol that cloud service customers can use to query a cloud service provider (CSP) on current security attributes related to a cloud service such as the current level of availability of the service or information on the last vulnerability assessment. This is normally done through a classical query-response approach driven by the customer. CTP also enables customers to define "triggers" in order to receive alerts when specific conditions are met, through a standard XMPP-based notification mechanism. The CTP API additionally provides access to a log facility that can be used to store and access security events generated by triggers.

It is important to emphasize that CTP mainly proposes a unified standardized API to present measurement results related to cloud security. As such, the CTP API does not cover the actual monitoring infrastructure and related technologies that are used to gather, store and analyze events in order to produce these measurement results.

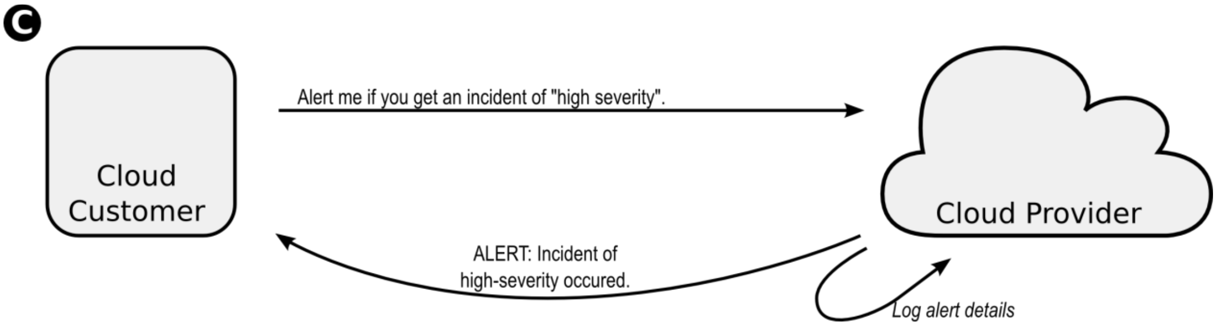
The following diagrams provide a general idea of the principles of CTP through 3 simple use cases where a cloud service customer uses CTP to query a cloud service provider about security attributes of its services. In the first figure (**A**), the cloud service customer uses CTP to query a cloud service provider about the service availability level that it is committed to provide. In CTP the result of this query is called an "objective" — or "service level objective" — since it describes what the provider aims to achieve, as typically described in an SLA.


A
Cloud Customer
What service availability level are you commited to achieve?
We are commited to at least 99% availability, calculated over a month.
Cloud Provider

Next, in figure (**B**), the cloud service customer queries the cloud provider about the service availability level that was actually achieved in the past month. The result of this query is called a "measurement result" in CTP, since it describes the result of a service level measurement reported by the cloud provider. Both this measurement result and the objective in the previous example apply to the same security attribute informally called "availability" here.


B
Cloud Customer
What service availability level did you reach in the past month?
Our availability was 99.2349% in the past month.
Cloud Provider

Finally, in figure (**C**), the cloud service customer asks the cloud provider to send an alert when a specific condition is verified. This is called a "trigger" in CTP. In addition, the cloud provider will also log the details of this alert locally for future consultation by the customer.


C
Cloud Customer
Alert me if you get an incident of "high severity".
Cloud Provider
ALERT: Incident of high-severity occured.
Log alert details

Naturally, for simplicity, these examples leave out a lot of details that are addressed in the specification.

2.2.2 THE CTP DATA MODEL

The CTP API structures data mostly in a hierarchical structure, where each customer has access to:

- **Service views**, which each refer to a particular individual cloud service, and are broken down into one or more
  - **Assets**, which refer to the components of the service (APIs, Databases, CPU, etc.), each having a set of distinct
    - Security **attributes**, which refer to measurable characteristics of an asset, and have one or more
      - **Measurements**, which refer to a specific process of evaluating an attribute, which is tied to a:

- A **measurement result**: the result/value of applying the measurement to the attribute.
- A **measurement objective**: a Boolean expression that describes the commitment the Cloud Service provider is making about the *measurement result*, as described in a SLA.
- An optional set of "**triggers**": conditions that create a notification to the customers, expressed as Boolean expressions just like *measurement results*.

Below is a simplified example of the above hierarchy applied to a cloud service called "Cloud Inc.":

- The customer Alice has a *view* that has one *asset*: A virtual machine.
- The virtual machine has one security *attribute*: "availability"
- The *attribute* "availability" has one *measurement* method called "monthly uptime" that provides
  - A *result*: x = 99.9834% of uptime in the past month.
  - An *objective*: x must be greater than 99.95%.

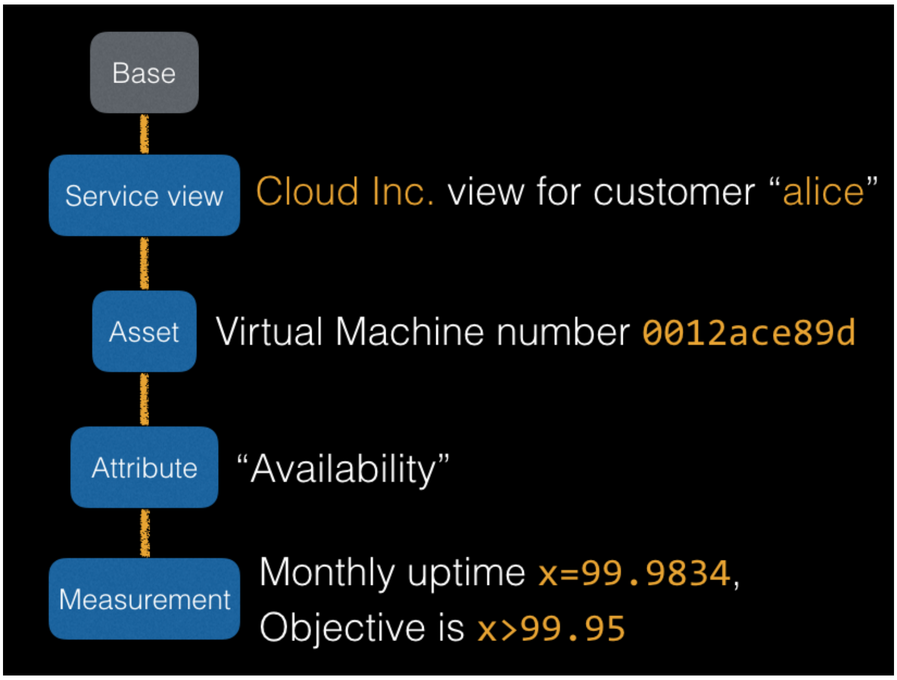

Base
Service view
Cloud Inc. view for customer "alice"
Asset
Virtual Machine number 0012ace89d
Attribute
"Availability"
Measurement
Monthly uptime x=99.9834,
Objective is x>99.95

Figure 2-3 Example of the hierarchy applied to a cloud service called "Cloud Inc."

Each element in the CTP Data Model is represented by an individual RESTful resource. With the exception of the optional triggers, the customer accesses these resources in read-only mode, through GET requests. Triggers are the only element that is configurable by the customers (they are created with a POST request).

Objectives and triggers in CTP both rely on Boolean expressions that test whether a certain measured value verifies a certain condition. These expressions are described with CTPScript: a language that is modeled after JavaScript expressions, with many simplifications that are designed to facilitate implementation. We highlight in particular that CTPScript is limited to statements representing expressions and does not include any other language construct (such as assignments, control structures, declarations, prototypes, etc.).

For example, a CTPScript expression that describes the fact that TLS symmetric key must be greater or equal to 128 bits might look like this:

```
value[0].tls_symmetric_key_length >= 128
```

More complex expressions are possible:

```
value[0].business_hours_uptime > 99.5 && value[0].other_hours_uptime > 98.0
```

2.2.3 THE CTP SERVER

The CTP Server implements the “CTP API and Data Model” (a.k.a. the CTP API): a JSON RESTful API that allows cloud customers to query cloud providers about the security level of their service. This is not enough however: the CTP API is mostly a “read-only” API and it does not describe how the CSP populates the data that is made available to the customer.

To create the CTP Sever and provide the means to populate the data that is made available to the customer, we simply extended the “read-only” CTP API with “write” operations and a tag based access control system. In other word, most API methods in the CTP API that are based on an HTTP GET were complemented with HTTP POST/PUT methods. This approach makes the CTP Server more technologically neutral. This non-official extension of the CTP API is described in the “CTP back-office API” specification[3].

Access control on the CTP server is based on a tagging mechanism. Briefly speaking, each user has a set of associated tags and each resource has a set of associated tags. Access is granted if the tags associated with the user match the tags associated with the resource being accessed. This approach allows fine grained access control: for example, a vulnerability scanner can be

---

[3] http://htmlpreview.github.io/?https://github.com/cloudsecurityalliance/ctpd/blob/master/client/CTP-Admin-API.html

granted permission to modify only the results associated with vulnerability measurement without any possibility to otherwise modify other resources on the CTP server.

The CTP Server is written in Go and relies on a MongoDB backend for scalability purposes.

2.3 STARWATCH

STARWatch is a SaaS application to help organizations manage compliance with CSA STAR (Security, Trust and Assurance Registry) requirements. STARWatch delivers the content of the Cloud Controls Matrix (CCM) and Consensus Assessments Initiative Questionnaire (CAIQ) in an online editable format, enabling users to manage compliance of cloud services with CSA best practices.

2.3.1 STARWATCH "VERSION 1"

The CSA CAIQ is a compliance questionnaire with 295 questions derived from the 136 control objectives defined in the CCM. Until recently, this questionnaire was typically answered through a standardized Excel spreadsheet, a cumbersome process that lacks desirable features such as track change or sharing functionalities. StarWatch was created as an online SaaS tool that allows filling in this questionnaire in a more agile way. In addition to replacing a spreadsheet, the tool offers the following additional functionalities:

- Track changes to the answers provided to a question.
- Work in a team; assign permissions to read/write assessments to users.
- Rate the relevance of a control as well as the maturity of the implementation.
- Cross-match CCM controls with over 30 different other standards; show only controls that map to a specific standard or even to a subsection of that standard (e.g., PCI-DSS, ISO/IEC 27001).
- Import existing assessments (in Excel) from a choice of over 200 questionnaires provided by major cloud providers around the globe.
- Export back to Excel when needed.

2.3.2 THE FUTURE OF STARWATCH

StarWatch is an evolving tool that CSA wants to adapt to the requirements of the cloud security community, including requirements that will emerge in the EU-SEC project. Several ideas are already being considered, such as:

- User defined questions.

- Filtering of questions/controls based on user assigned relevance.
- Comparisons between assessments based on user-selected criteria.
- Annotate controls/questions with reference to evidence.
- Etc.

More importantly, CSA is considering extending StarWatch with a "continuous" self-assessment mode.

- Users would declare a self-assessment as "continuous" by associating it with a policy. The Policy itself would define:
  - SQOs expressed as commitments to the implementation of CCM controls or CAIQ questions.
  - Update requirements for each SQO (e.g., once per week).
- A policy engine would then monitor updates made to the self-assessment for discrepancies and produce a "certification status" reflecting the self-assessments.

2.4 SLIPSTREAM

SlipStream[4] is a multi-cloud application deployment engine and brokerage system that federates any number of clouds and allows users to deploy and manage cloud applications on and across those clouds (Figure 2-4). It is the central management and control behind Nuvla, the SaaS deployment of SlipStream that is managed by SixSq, which is the central access point for users' cloud resources.

Through Nuvla[5], users can easily automate the deployment and maintenance of their platform, targeting any connected cloud without having to change the application definition.

---

[4] https://sixsq.com/products-and-services/slipstream/overview

[5] https://sixsq.com/products-and-services/nuvla/overview

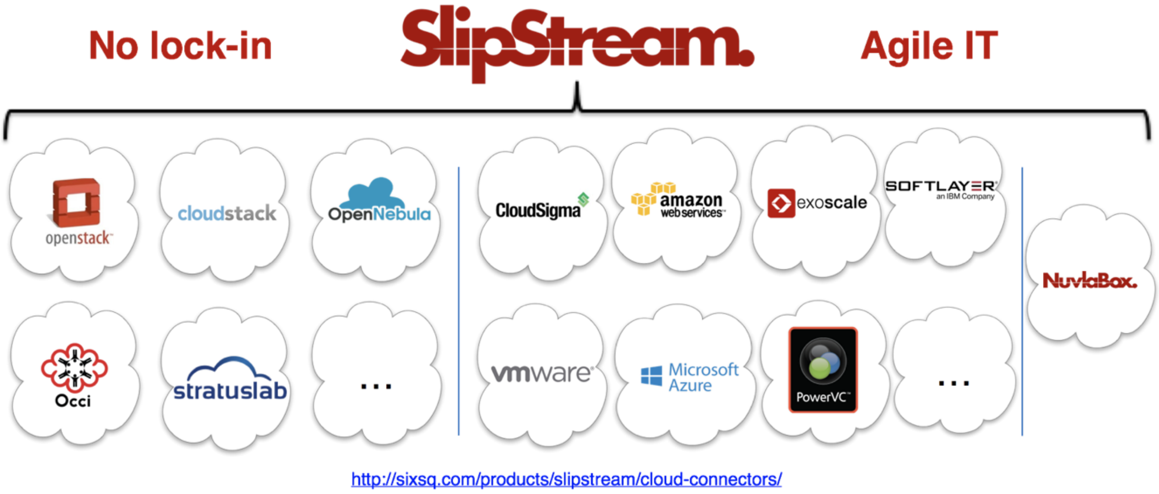


Figure 2-4 High level overview of the multi-cloud application management offered by SlipStream

Leveraging resources from Infrastructure as a Service (IaaS) cloud providers, SlipStream manages cloud applications through the full lifecycle: deployment, configuration, validation, scaling, and termination (Figure 2-5).

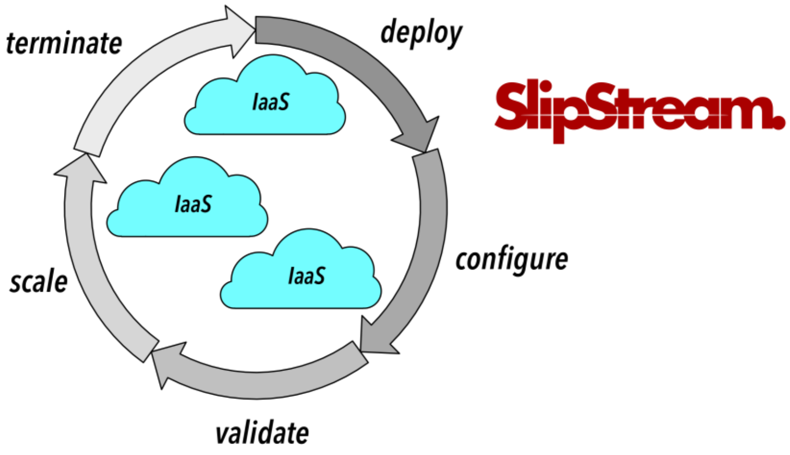


Figure 2-5 Full application lifecycle management through SlipStream

SlipStream's essential features include:

- Enterprise App Store built-in: Self-service IT delivered for the enterprise, simplifying application provisioning dramatically;
- Recipe/template/blueprint: Define and execute deployments, based on high-level recipes (script, Puppet, Chef, Ansible, etc.);
- Cloud Broker Enablement: Supports most public and private IaaS;
- Multi-cloud Management: Supports hybrid and multi-cloud deployment scenarios.

2.4.1 USERS AND BENEFITS

Cloud technologies provide real benefits to users and organizations, but they also have their own challenges.

- Incompatible APIs: Make it difficult to move applications from one cloud to another and complicate the simultaneous use of different clouds.
- Opaque VMs: Keeping track of what virtual machines contain (data and services) and managing their updates are difficult.
- Component vs. Application: Most applications comprise multiple layers with numerous individual machines. Cloud services oriented towards single VMs make application management more tedious.

SlipStream addresses these challenges by providing its users with an efficient platform for the management of the full lifecycle of cloud applications.

A number of different types of people within an organization can benefit from SlipStream:

a) Those who are working on different projects and need IT applications and resources – they can benefit from the SlipStream App Store where they can start the applications they need with one click;
b) Those who manage a number of workers taking advantage of cloud resources and want an overview of their resource usage to understand costs and their involving needs – SlipStream provides the ability to monitor resource utilization;
c) Those who develop cloud applications for other people within their organization – they benefit from SlipStream by creating a rich catalog of services that can be automatically and reliably deployed; and
d) Those who manage their own SlipStream installation – they can integrate their own cloud infrastructure into their SlipStream deployment and control what external cloud resources are available to their users.

Read more about possible SlipStream use cases at http://sixsq.com/products/slipstream/usecases/.

2.5 TOOL CHAIN: INTERACTION BETWEEN COMPONENTS

Figure 2-6 shows a high-level overview of the interaction between tools which are either already existent (see previous sections) or will be developed in the course of EU-SEC.

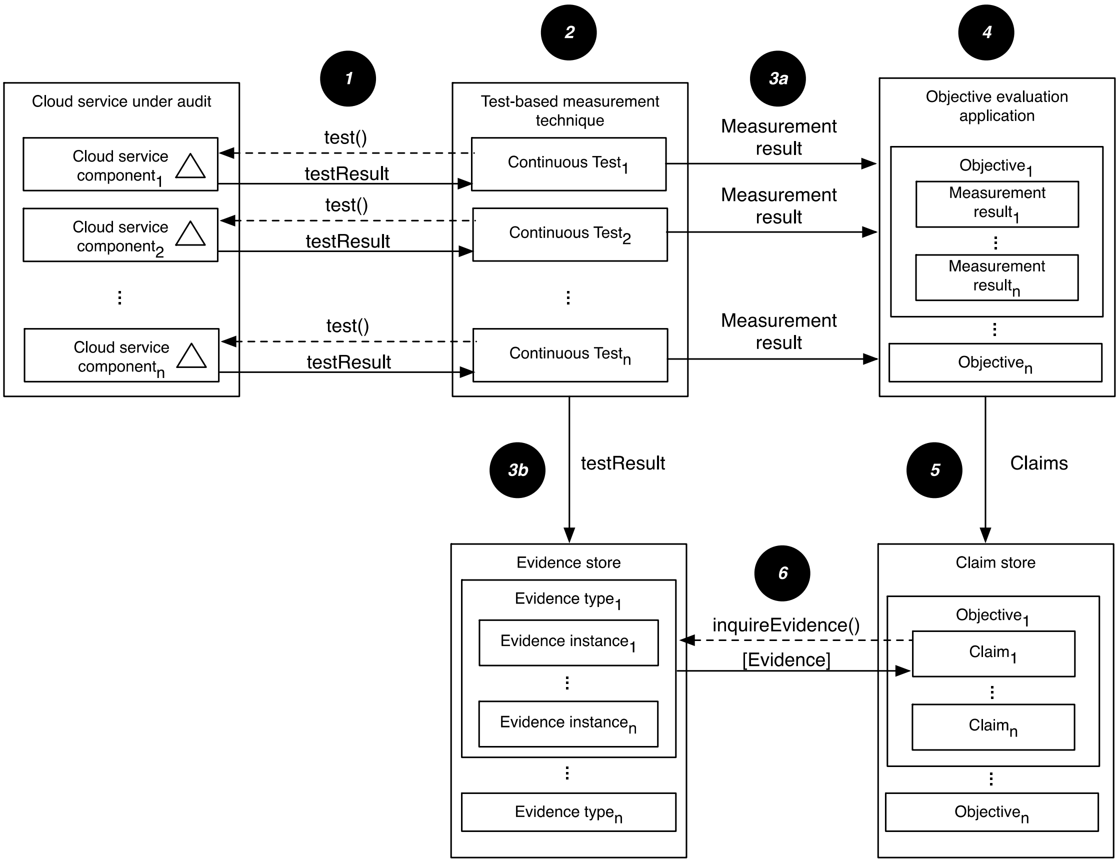


Figure 2-6. High-level view on tool interaction

The interaction between the tools works as follows: A continuous test-based measurement technique such as Clouditor uses tests to produce *evidence* (Step 1). Each test result is stored in the evidence store (Step 3b) where it can later be looked up by a customer or auditor in case of, e.g., disputes. This point will be further detailed in Step 4. Note that only parts of a test result are considered evidence whereas the test result already embodies a decision made on the basis of the information which has been obtained during the test's execution[6]. The test-based measurement technique applies some function which is referred to as *test metric* to the test results which it observes, e.g., counts the occurrence of failed tests or the duration of successively failing tests (Step 2, for further details see also Chapter 4 of Deliverable 3.2). The output of that function is referred to as *measurement result*. These measurement results are supplied to the objective evaluation application (see Step 3a) which uses rules to reason about the measurement results, e.g., according to the measurement results, has the cloud service been available for at least 99.999% during the last 360 days (Step 4). The CTP API provides the

---

[6] In terms of testing terminology, any information which serves as input to well-defined test oracles are considered evidence. This renders each test oracle which forms a part of a test a primitive metric (for further information see Section 4.1.5, Deliverable 3.2).

specification to implement such an objective evaluation application. The result of applying these rules determines whether a cloud services satisfies a particular control objective derived from some control of a certification scheme. Note that this mapping is based on manually derived expert consensus, i.e., there is no rigorous method available to automatically interpret a control objective. The result of evaluating a control objective is referred to as a *claim* stating either a controls satisfaction or dissatisfaction at a certain point in time. The claims are forwarded to the claim storage where they are persisted (Step 5). In case an authorized party, e.g., a cloud service user, has doubts about the claim or wants to confirm the claim, the customer can inquire the evidence (contained in the atomic the test results) which was used to generate the claim (Step 6). Deployment and management of applications involved in the tool chain, e.g., evidence store and claim store, can be facilitated through SlipStream.

3 INTEGRATING CONTINUOUS SECURITY AUDITS

This chapter describes the steps involved when integrating the tool chain described in the previous chapter with an existing cloud service. The following section provides a high-level overview of the process while Section 3.2 describes each step of the integration process in detail.

3.1 OVERVIEW

Figure 3-1 shows the steps which need to be taken to integrate the tool chain described in Section 2.5 with a cloud service which is sought to be subject to continuous security audits. These steps include:

1. *Select global integration strategy for toolchain:* In the first step, the general integration strategy for the toolchain is selected which is driven by the additional risk which a cloud service provider is willing to tolerate when planning to support continuous security audits.
2. *Deploy tool chain:* Drawing on the general integration strategy, in the second step, the deployment strategy is determined, that is, it is defined where to run certain parts of the continuous security audit tool chain, including: Test-based measurement techniques, objective evaluation, as well as evidence and claim storage. Since the deployment strategy of the tool chain is derived from the global integration strategy, deployment of the tool chain is also risk-driven.
3. *Discover cloud service:* In the third step, the components of the cloud service which is sought to be subjected to continuous security audits are discovered.
4. *Derive feasible measurement techniques:* In the fourth step, feasible evidence production techniques for the discovered cloud service are derived.
5. *Select feasible metrics:* In the fifth step, the measurement results are derived based on the evidence that can be produced for a discovered cloud service.
6. *Start execution of measurements:* In the sixth step, the execution of the measurement techniques is triggered, thereby rendering the tool chain operational.
7. *Adapt measurement techniques at operation time:* In the seventh step, compositional as well as configuration changes of the cloud service under audit are continuously

discovered at operation time of the tool chain. In case of changes, evidence production techniques are adapted accordingly while preserving semantics of computed measurement results.

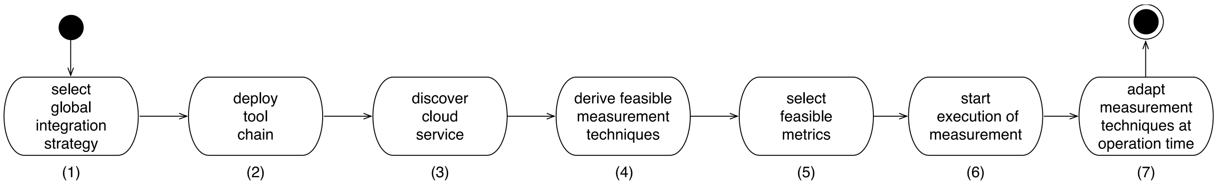

select global integration strategy
(1)
deploy tool chain
(2)
discover cloud service
(3)
derive feasible measurement techniques
(4)
select feasible metrics
(5)
start execution of measurement
(6)
adapt measurement techniques at operation time
(7)

Figure 3-1: Integration process of tool chain to support continuous security audits of cloud services

3.2 INTEGRATION PROCESS

This section describes the steps of the integration process in detail.

3.2.1 STEP 1: SELECT GLOBAL INTEGRATION STRATEGY

This step determines the global integration strategy of the tool chain.

Note that the discussion of integration variants described hereafter relies on the following assumption: Integrating parts of the tool chain which do *not* directly interact with the cloud service under audit (i.e., evidence store, claim store and objective evaluation application) as part of the service's infrastructure provides superior security properties. The rationale behind this is that adding further external environments to run parts of the tool chain leads to a relatively higher increase in attack surface because these other external environments (i.e., infrastructure where tool chain parts can be run) have be communicated with as well as maintained in a secure manner. However, it is important to point out that this assumption does not always have to be true, for example, if the cloud service provider under audit is malicious and attempts to manipulate parts of the tool chain to alter, e.g., measurement results.

RISK-DRIVEN INTEGRATION OF MEASUREMENT TECHNIQUES

Different levels of invasiveness are introduced hereafter which a continuous security audit tool may require to produce evidence as well as measurement results to support the validation of security controls. Recall that a continuous security audit tools can draw on two classes of measurement techniques: Monitoring-based and test-based measurement techniques. The former use monitoring data as evidence which is produced during productive operation of a cloud-service. The latter also collects evidence while a cloud-service is productively operating. Different to monitoring-based methods, however, test-based methods do not passively monitor operations of a cloud service but actively interact with it through tests.

The level of integration required for evidence and measurement result production is determined by the changes of the productive environment of the cloud service to be continuously audited, that is, the required changes of each component involved in productive service delivery. Hereafter, *non-invasive*, *minimally invasive* and *invasive* integration of measurement techniques are described.

- *Non-invasive integration*: As the name indicates, this type of integration requires no change of the productive environment which is used to operate the cloud service under audit. This means that a measurement technique can produce suitable evidence without requiring any changes to the cloud service. This type of integration implies that the implementation of the measurement technique does not have to be part of the cloud service infrastructure but can operate on a remote host, external to the cloud service's infrastructure.
  As a basic scenario, consider the endpoint of a SaaS application, i.e., a web site which is publicly reachable. In order to automatically produce measurement results as to whether this endpoint supports secure communication with its users, no further privileges are needed. As a different example, consider a SaaS application to which only authorized user have access. In order to automatically assess whether, for example, any input fields available to authorized users properly validate user input and thus do not possess some SQL injection vulnerability, user level access privileges are required. Still, this example measurement technique does not require to change the composition or configuration of production environment of the cloud service.
- *Minimally invasive integration*: This type of integration requires to change the configuration of the production environment of the cloud service under audit to permit the measurement technique to produce measurement results. Similar to non-invasive integration, minimally invasive techniques does not have to be deployed and operated as part of the cloud service's infrastructure.
  As an example, consider changing security groups to allow a remote host sending TCP segments to a cloud service component, e.g., a virtual machine to check its responsiveness. The original security model of the cloud service may not permit some components to be accessed from external hosts which are not part of the cloud service's

infrastructure. Therefore, in this example, the configuration of the cloud service under audit has to be altered for the measurement technique to work correctly.

- *Invasive integration*: This type of integration requires to change the composition of or the applications used by a cloud service's productive environment to allow measurement techniques to produce suitable measurement results. Contrary to non-invasive and minimally invasive integration, invasive integration of measurement techniques implies that at least some parts of techniques' implementation are integrated with the production environment which is used to operate the cloud service under audit. We can distinguish the following subtypes of invasive integration:
    1. *Compositional changes:* In this case, structural changes to the cloud service composition are needed such as adding a virtual machine or micro service where the measurement technique is deployed and operating on. A classic example of invasive integration through compositional changes are so-called *monitoring agents*, i.e., additional applications deployed on virtual or physical components of the cloud service collecting information such as CPU load.
    2. *Code-level changes:* Here, changes in the form of patches to applications which constitute components of cloud services are needed in order to produce measurement results. Consider, as an example, changing the scheduler of a cloud platform management system such as OpenStack to be able monitor deployments of virtual machines to determine if some machines of a particular user are only using designated hosts, that is, do not to share the underlying hardware with machines of other users.

Changing configuration (i.e., minimally invasive integration) or composition (i.e., invasive integration) of the production environment of the cloud service to be continuously audited may increase the attack surface of the service. Therefore, selecting a suitable integration strategy is driven by risk assessment of the cloud service provider whose service is subject to continuous audit.

EXAMPLE

Let's assume that the cloud service provider is only willing to subject her cloud service to non-invasive integration of measurement techniques. The reason for this choice is that the risk assessment of the provider has determined that the additional risk entailed with minimally invasive as well as invasive techniques is *not* tolerable. This implies that presumed benefits of increased transparency provided by continuous security audits are outweighed by the additional risks incurred by configuration and compositional changes.

Let's be more specific and assume that a provider considers non-invasive integration of a measurement technique to check if communication with his public service endpoints via insecure networks is configured in a secure manner. Through establishing a connection to the endpoint, the desired technique determines if SSL/TLS configuration of a cloud service's web server allows to securely communicate with the service. To that end, the technique uses a metric to compute a measurement result, that is, a score indicating the strength of the configuration. The underlying model to compute this cipher suite score takes into account known SSL/TLS vulnerabilities such as OpenSSL Heartbleed, CRIME or OpenSSL CCS Injection. Also, the web server must not support TLS fallback signaling cipher suite value (scsv) and secure session renegotiation. Lastly, the web server must not accept self-signed certificates.

The question at this point is: What residual risks does using such a non-invasive measurement technique entail? Let's first consider the evidence which needs to be produced in order for this technique to calculate measurement results. One example parameter of the technique's metric is whether the endpoint supports self-signed certificates. Since we are considering a publicly exposed endpoint, this information is public as well, that is, potentially anybody can determine that the endpoint supports self-signed certificates. The same applies to the remaining evidence produced by the measurement technique. This means that anybody may produce the required evidence and compute the cipher suite score. Therefore, one may argue that using this non-invasive measurement technique does not pose any additional risks.

RISK-DRIVEN INTEGRATION OF EVIDENCE STORE

Recall that the evidence store is responsible for persisting produced evidence for some predefined period of time. Consequently, the evidence store inherits the challenges of overexposing critical information contained in the evidence as well as protecting evidence against unauthorized alterations (see also Section 2.4.1 of Deliverable 3.3). Therefore, a risk-driven integration of the evidence store is needed, that is, the risk of disclosed, altered or deleted instances of evidence has to be assessed to determine whether the evidence store is integrated as part of the infrastructure of the cloud service under audit or external to the service's infrastructure, on a remote host. Further, in order to decide how to integrate an evidence store, the additional risk of producing any evidence of any measurement technique using that particular store to persist evidence has to be considered. From the perspective of a cloud provider, the global, additional risk exposure will be determined by the highest additional risk incurred by producing some type of evidence.

EXAMPLE

Recall our example of a non-invasive measurement technique which connects to a cloud service's endpoints and, based on this evidence, computes a cipher suite score. Here, evidence consists of, e.g., the information that an endpoint possesses some known SSL vulnerability or supports self-signed certificates. As already discussed in the previous paragraph, evidence obtained from this measurement technique is public if the endpoint is publicly reachable.

In context of the integration of the evidence store, this risk exposure is further affected by an evidence store instance which is shared by multiple measurement techniques which are producing evidence for the cloud service. Consider, for example, also storing evidence indicating SQLI vulnerabilities of the cloud service's web application components. With regard to the evidence store deployment, the question is now – given both types of evidence – what is the global, additional risk exposure? Answering this question, again, depends on the individual risk assessment of the cloud service provider which determines whether to integrate the evidence store as part of the cloud service's infrastructure or externally.

Note that an evidence store may be shared between multiple cloud service providers, that is, between multiple measurement techniques producing evidence for multiple cloud services and providers. This case can lead to an increase in risk because a successful attack may disclose evidence produced for multiple cloud services of different providers.

RISK-DRIVEN INTEGRATION OF OBJECTIVE EVALUATION APPLICATION

As described in Section 2.5, the objective evaluation application consumes measurement results and, on this basis, reasons about SLOs and SQOs where the outcome of that evaluation is referred to as claims. Both measurement results as well as claims possess a higher level of abstraction than the evidence used to compute the measurement results. Naturally, a strong separation of evidence and measurement results has to be ensured, that is, results forwarded to the objective evaluation application must not contain any evidence used to compute the respective measurement results. Yet, despite a higher level of abstraction, measurement results' evaluation may still leak information to unauthorized parties, i.e., has a SLO or SQO been satisfied or not. Thus, the additional risk incurred if these results are forwarded to an objective evaluation application not part of the infrastructure of the cloud service under audit has to be assessed.

Note that there may not exist any unauthorized parties if the measurement results and claims are considered to be publicly accessible. In this case, there is no potential damage and thus no risk to consider when forwarding results from the continuous measurement technique to a remotely hosted objective evaluation application. Otherwise the evaluation application can also be integrated as part of the infrastructure of the cloud service under audit. Note that in the latter case, it is reasonable to expect that the measurement techniques are also integrated in a minimally invasive or invasive manner. Otherwise, evidence produced by the technique as well as computed measurement results exist outside the cloud provider's infrastructure already.

EXAMPLE

Consider, for example, measurement results which indicate whether any persistent storage of the cloud service is encrypted (and only decrypted as needed, e.g., if a query is issued to retrieve some data). To that end, evidence regarding the various types of storage a cloud service may employ, e.g., object storage, relational databases and so forth, has to be produced. This evidence is then provided as input to a suitable metric computing the measurement result at some point in time. This metric may only output a result such as *StorageIsEncrypted* or *StorageIsNotEncrypted*. In this case, it is obvious that if these measurement results were to be disclosed to an unauthorized third party – due to, e.g., vulnerabilities in the objective evaluation application – the potential damage regarding an attacker seeking to cut corners in his attack vector is relatively small since the information obtained is limited.

RISK-DRIVEN INTEGRATION OF CLAIM STORE

A *claim* refers to the result of evaluating a control objective stating a control's satisfaction at a certain point in time. In order to determine whether a control objective is satisfied, one or more measurement results are necessary. A claim is established by the objective evaluation application and then forwarded to the claim store for persistence. The claim store is either part of the infrastructure of the cloud service under continuous audit or hosted on a remote host, external to the service's infrastructure.

A claim allows deriving what type of measurement result was used to establish the claim. However, it does not tell us anything about the underlying model of the measurement result, that is, the metric which was used to compute the result. Therefore, we cannot directly infer which evidence lead to establishing the claim.

Yet the history of claims may permit conclusions if a control objective is dissatisfied. This, in turn, can translate into time savings on an attacker's side because the attacker – if a claim's history is disclosed by unauthorized parties – may filter for potential security issues by absent claims previously satisfied.

The above considerations guide the decision how to integrate a claim store: If disclosing the claim history is considered an intolerable risk, then the claim store can be integrated as part of the cloud service under audit. Note that this does not imply that the objective evaluation application establishing the claims in the first place is also integrated as part of the service's infrastructure. The reason for this is that the evaluation application does *not* store any computed claims longer than evaluation requires.

EXAMPLE

Consider the claim *During the last 24 hours, the TLS configuration of a cloud service's endpoint was secure*. Let's assume this claim has been reissued for some time, e.g., 10 times in succession, suddenly coming to a halt, that is, no such claim is forwarded to the claim store anymore. The absence of such a claim may indicate that the service's endpoints are not securely configured. This, in turn, can serve as a starting point for an attacker who gained access to the claim store and intends to attack the cloud service under audit.

3.2.2 STEP 2: DEPLOYMENT OF TOOL CHAIN

In this step, the continuous security audit tool chain introduced in Section 2.5 is deployed. To that end, it is first necessary to determine the deployment strategy for the tool chain, i.e., where to run certain parts of it. The deployment strategy is derived from the global integration strategy described in the previous section. To that end, each component's planned integration is inspected and, on this basis, it is determined where to deploy the respective component. Note that although each component of the tool chain can, in principle, be deployed at a different location, it is reasonable to expect that such a fully distributed tool chain is undesired due to various reasons, e.g., performance, reliability and security considerations.

Once deployment of the tool chain is completed, all required components of the tool chain are installed at their desired location. Note that the tool chain is not yet operational since no concrete measurement techniques have been select which, in turn, depends on the components the cloud service consists of. Determining which measurement techniques are feasible and, on this basis, which suitable metrics to select will be described in the next three steps of the integration process.

3.2.3 STEP 3: DISCOVER CLOUD SERVICE

In order to determine which measurement techniques can be used in context of a concrete cloud service instance, this step discovers a cloud service including, e.g., applied security policies, components and exposed network services. To that end, complementary discovery techniques are used which assemble available information about a cloud service under audit into a so-called *service description*. A service description, therefore, can be understood as a summary of components and configurations which constitute a particular cloud service.

An instance of a service description is derived from a general model to describe cloud services. An extract of that general model is shown in Figure 3-2 which has been developed on the basis of OpenStack. Depending on the cloud service to be audited, the general cloud service description model is extended, for example, by adding descriptions for specific services provided by Microsoft Azure.

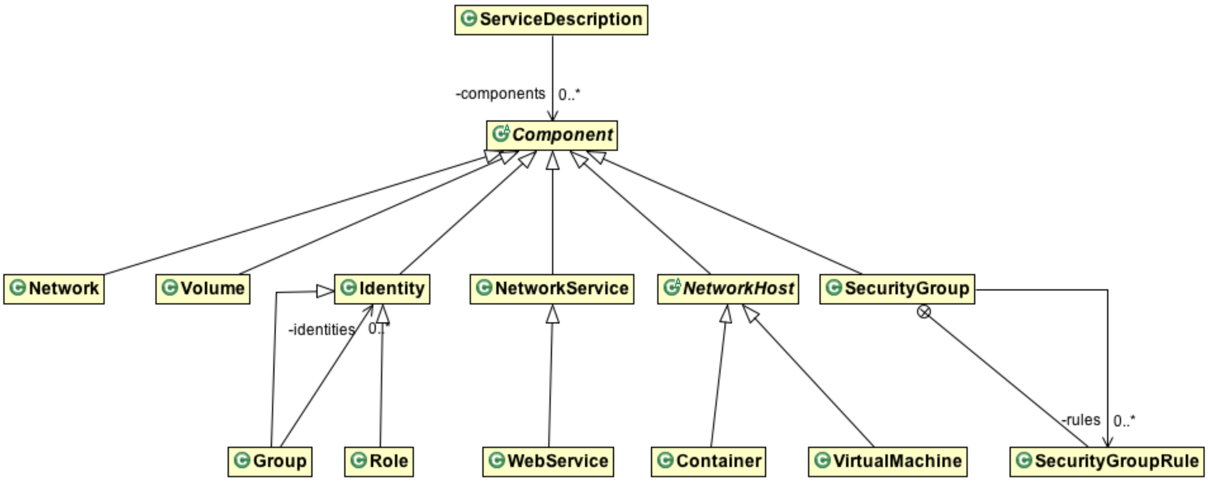


Figure 3-2: Extract of service description for IaaS provided by OpenStack

It is obvious that the scope of a generated service description depends on the access privileges which discovery techniques are granted by the cloud service provider. These privileges, in turn, result from the global integration strategy selected in the Step 1. Put differently: Discovery techniques are integrated in the same way as are measurement techniques, thus having the same privileges to access components of the cloud service to be audited.

Consider, for example, a cloud service provider only having agreed to a non-invasive integration strategy where, as a consequence, the measurement techniques have to be deployed external to the infrastructure of the cloud service. Given this integration strategy, the discovery techniques also can only discover a cloud service in a non-invasive manner, e.g., through scanning its exposed services using tools such as Nmap[7]. In contrast, when assuming that the provider has agreed to minimally invasive integration, then a discovery technique may be assigned a specific user (e.g., AWS's security auditor role) with whom it can call admin APIs of the cloud service under audit and retrieve more detailed information about the infrastructure of the cloud service.

EXAMPLE

Recall that the selected integration strategy for measurement techniques in our example scenario is non-invasive. Therefore, discovery techniques which can be used to assemble a service description are confined to only interacting with the cloud service's interfaces, without having privileges to enforce configuration changes (minimally invasive integration) or compositional changes (invasive integration) of the cloud service's infrastructure.

Let's assume that as one result of non-invasive discovery, any (publicly reachable) HTTPS endpoint of the cloud service is discovered. More specifically, part of the service description assembled by the discovery techniques contains all publicly reachable IP of hosts which expose port 443, the default port used by HTTPS.

3.2.4 STEP 4: DERIVE FEASIBLE MEASUREMENT TECHNIQUES

In this step, *feasible* measurement techniques are derived by matching the information obtained from the service discovery with the set of all available measurement techniques provided by the tool chain. Thus, feasible measurement techniques denote those techniques which can be actually used in context with a concrete cloud service instance.

In order to identify feasible measurement techniques, the *preconditions* for each technique have to be identified and modelled as constraints, i.e., a set of rules which has to be satisfied. These preconditions represent assumptions about the environment the technique is operating in as well as the input required by the technique such that it produces complete and correct (i.e., as specified) measurement results. This means that a particular measurement technique can only be used if the cloud service to be continuously audited fulfills the preconditions of the technique.

In the case of test-based measurement using tools such as the Clouditor, the preconditions can be partly derived from the continuous test configuration written in *ConTest* (see Deliverable 3.2): The input parameters specified for each test cases (the primitive of each continuous test-

---

[7] https://nmap.org/

based measurement technique, see Section 4.1.2 of Deliverable 3.2) provide some indication as to what the technique assumes about the environment of the cloud service under audit. As an example, consider having an input parameter *hostname* which suggests that the evidence production technique expects a host which can be reached over an IP-based network. Naturally, assigning semantics to input parameters has to be conducted manually per test case.

Yet inspecting test cases of the test configuration alone does not suffice when eliciting the preconditions of a measurement technique. Additional constraints have to be considered, e.g., the security group granting a remote host – where the technique may be deployed in case of non- or minimally invasive deployment – access the cloud service component to be audited. This is where the service descriptions obtained in the previous step come in: In order to check if such additional constraints are satisfied, additional information about the cloud service have to be available through the service description.

To summarize: In order to derive feasible measurement techniques, the preconditions under which an evidence production technique will work correctly are modelled as a set of rules. These rules draw on the information provided by the service descriptions to check if a particular evidence production technique can be used in context with a concrete cloud service instance.

Lastly, some measurement techniques – even though technically feasible – might not be used at all due to operational risks. This is the case if a technique will foreseeably lead to a significant increase of operational costs of the cloud service infrastructure. Consider, for example, a technique which measures the available bandwidth of a cloud service component where measurement results are used to check whether the available bandwidth is sufficiently high to prevent certain types of Distributed Denial of Service (DDoS) attacks. Furthermore, next to risks originating from increasing operational costs, additional risks can result from the possibility of a measurement technique unintentionally disrupting regular service operation. In this case, the risk consists of a financial loss which is incurred in case of service downtime.

EXAMPLE

In order to illustrate the derivation of feasible measurement techniques, consider the following example scenario: Let’s assume that three measurement techniques are available which produce evidence to check which TLS cipher suites an endpoint is using to secure communication via HTTP (i.e., HTTPS). The first technique inspects the configuration used by the webserver which defines the TLS configuration, e.g., accepted cipher suites. The preconditions of this technique require that it has access to the virtual machine where the webserver is running and has sufficient privileges to read the webserver’s configuration file. The second technique obtains the required evidence by connecting to the endpoint and inspecting what cipher suites are offered by the TLS endpoint at runtime. In order for this technique to work correctly, it has to be able to reach the host exposing the HTTPS endpoint and start a TLS connection, that is, conduct a TLS handshake. The third technique inspects log files generated by the web server and, provided a sufficiently detailed log level, retrieves accepted cipher suites from the log. Similar to the preconditions of the first technique, this technique requires sufficient privileges to access the webserver's log data. Note that these log files may not only be available at the host where the webserver is running, but also be forwarded to a central logging system using tools such as logstash[8] permitting operational monitoring of a cloud service's endpoints.

Recall that in the previous section, it was assumed that part of the service description assembled by the discovery techniques contains a publicly reachable IP of a host which exposes port 443. Given this exemplary extract of a service description, it can be concluded that the preconditions of the second evidence production technique are satisfied. This means that the second technique can be used with the example cloud service instance to produce evidence which allows to determine if an endpoint is securing communication via HTTP using strong TLS cipher suites.

3.2.5 STEP 5: SELECT SUITABLE METRICS

Having completed Step 4, we now know which specific measurement techniques can be used with a particular cloud service instance. Each measurement technique supports computation of measurement results according to one or more metrics. The question which this step addresses is which measurement results should be produced?

As described in Section 2.5, *measurement results* serve to evaluate service level objectives (SLO) or service quality objectives (SQO). Yet the problem is that measurement results used to evaluate a SQO or SLO – contrary to their name – cannot be directly measured because they already incorporate an abstraction, i.e., a *property model* necessary to allow to rigorously evaluate the respective objective. Thus, *measurement results* are understood as the output of a metric which takes as input the actual raw data, i.e., the *evidence* which has been obtained by some suitable evidence production technique and, on this basis, performs a predefined computation, thus determining the value of the measurement result.

As laid out in Deliverable 1.4, a SQO is "the commitment a cloud service provider makes for a specific, qualitative characteristic of a cloud service, where the value follows the nominal scale

---

[8] https://www.elastic.co/products/logstash

or ordinal scale (2).Further, a SLO is defined as "the commitment a cloud service provider makes for a specific, quantitative characteristic of a cloud service, where the value follows the interval scale or ratio scale (2). Thus, in order to determine whether a SQO or SLO is satisfied, test metrics have to be available which output *measurement results*.

- *Measurement results for SQOs:* Characteristics whose values are measured on the nominal scale or ordinal scale imply that reasoning about a SQO is confined to classification and comparison. Put differently: It is at least possible to state whether a cloud service possesses a particular characteristic (nominal level). Consider, as an example, the SQO "*User data persisted by the cloud service is encrypted*". Provided having proper measurement techniques available, the value of this characteristic at a certain point in time is either true or false. Further, if a cloud service's characteristic can be measured on an ordinal level, then measured values can be compared and sorted. For example, a SQO can state that the encryption algorithms used to encrypt sensitive data have to be highly secure. Given a suitable metric, values for this characteristic may be observed indicating *insecure*, *secure* and *high-secure* encryption algorithms where the strict order for these measured values is *insecure* < *secure* < *high-secure*. Intuitively, one may assume that – given the above scale – values observed for secure and high-secure encryption algorithms are somewhat more similar than values indicating insecure and secure algorithms. However, this is incorrect: Measuring on the ordinal scale does not provide any information about the distance between two ranks.
- *Measurement results for SLOs:* Measuring values on the interval as well as on the ratio scale allows to make statements about the difference in measured values. As an example, consider the SLO "A vulnerability of a cloud service has to be fixed within 8 hours after discovery." Let's assume that a suitable measurement technique exists which produces the required evidence to compute the desired measurement results allowing to reason about this SLO, e.g., the minutes it took to fix a discovered vulnerability. This measurement result follows an interval scale since the units on the (time) scale are equal to each other, i.e., the difference between 60 and 120 minutes is the same as between 180 and 240 minutes. Further, time is a ratio scale since it possesses a meaningful zero point, thereby permitting comparisons such as fixing the last vulnerability took twice as long as fixing the preceding one.

At this point, it is important to note that it is assumed that a mapping between measurement results and SLOs and SQOs exists which has been agreed upon by domain experts in a prior effort. Having a mapping between measurement results and SLOs and SQOs available means that once feasible measurement techniques have been identified (Step 4), it can be deduced –

based on the feasible metrics these techniques support – which SLOs and SQOs of a concrete cloud service instance can be automatically audited. The selection of suitable metrics from those that are technically feasible then depends on the SLOs and SQOs according to which a cloud service shall be audited continuously.

EXAMPLE

Recall that in Step 4, a feasible measurement technique has been identified which obtains the required evidence by connecting to the endpoint and revealing what cipher suites are offered by the TLS endpoint at runtime. This evidence can serve as input to a set of test metrics which compute measurement results to reason about SLOs and SQOs.

In our example case, this function may inspect the TLS cipher suites offered by the endpoint to check if it only contains suites which are considered *strong*. These strong cipher suites are predefined in a whitelist, in accordance with the current state of the art. If the endpoint only accepts strong cipher suites, then one feasible metric may output the measurement result *isStrong*. If any other cipher suites are accepted, then the function outputs the measurement result *isNotStrong*. These measurement results follow the nominal scale since they indicate to which group the offered TLS cipher suites belong, that is, either they are all strong (*isStrong*) or they are not all strong (*isNotStrong)*.

An example sequence of measurement results obtained by repeatedly executing the evidence measurement technique and computing measurement results by applying the metric may look like this: <*isStrong*, *isStrong*, *isNotStrong*, *isStrong*>. Lastly, having these measurement results available, satisfaction of the following, example SQO can be evaluated: *Every communication channel between the cloud service and a client using HTTP over an insecure network is secured using strong TLS cipher suites*.

Let's consider another example of a feasible test metric which is based on the measurement technique which obtains TLS cipher suites supported by the cloud service's endpoints through connecting to them. In this case, measurement results returned by the test metric ought to indicate for how long a cloud service's endpoint supported one or more cipher suites which are considered insecure, i.e., are not strong. Put differently: The measurement results indicate how long it took the cloud service provider to fix a vulnerable TLS configuration. To that end, the test metric stores the time when it first encounters the cloud service's endpoint to support TLS cipher suites *not* considered strong; however, no measurement result is produced just yet. Only the next time when inspecting the evidence indicates that all accepted cipher suites are strong, i.e., the vulnerable configuration has been fixed, a measurement result is produced whose value follows the ratio scale stating the time (e.g., in seconds) it took to apply the fix. Naturally, this test metric is only feasible if each instance of evidence in this example case contains the time of creation.

An example sequence of measurement results obtained by repeatedly executing the evidence production technique and computing measurement results may look like this: *<123,345,44,514,78>*. Having these measurement results available, satisfaction of the following, exemplary SLO can be evaluated: *Insecure communication channels which results from misconfigurations have to be fixed within 480 minutes (or 8 hours) after discovering the vulnerable configuration.*

3.2.6 STEP 6: START EXECUTION OF MEASUREMENTS

Having selected suitable metrics to reason about SLOs and SQOs, the tool chain is put into operational state by triggering the execution of the measurement techniques required to compute the selected metrics.

3.2.7 STEP 7: ADAPT MEASUREMENT TECHNIQUES

Once the initial configuration of the tool chain has been deployed, an additional question is how to adapt to changes in composition as well as in configuration of the cloud service under continuous audit. Such changes may lead to deployed measurement techniques not working correctly anymore, thus not providing correct evidence to compute measurement results. Therefore, it is necessary to continuously check whether the preconditions of deployed measurement techniques are still satisfied. To that end, discovery techniques which are used to assemble service descriptions can be leveraged (see Section 3.2.3). More specifically, these discovery techniques are executed continuously at operation time of the tool chain to check if the information contained in the derived service descriptions still satisfies the set of rules, i.e., the preconditions of a deployed measurement technique.

In case the preconditions of a measurement technique are still satisfied, no further action to adapt the measurement techniques is needed. In case its preconditions are not satisfied anymore, however, this technique is no longer considered feasible. Thus, the operation of the now infeasible technique is terminated. This implies that measurement results which were computed using this evidence cannot be computed anymore and are thus not available to reason about the satisfaction of SLOs or SQOs associated with the measurement results.

Once an infeasible measurement technique has been terminated, the latest service description is then used to find alternative techniques whose outputs, i.e., measurement results are semantically similar. Note that since the discovery techniques are integrated with the same level of invasiveness as the measurement techniques, it is reasonable to assume that an alternative measurement technique – if existent – is technically feasible.

If such a feasible alternative technique is found, then the remaining question is which risks are incurred by different deployment variants of the alternative technique (similar to Step 2). This means that it is necessary to assess the risks associated with producing evidence and measurement results using the alternative technique. Since evidence instances are used as input to a at least semantically similar test metric, the evidence produced by the alternative technique has to be somewhat similar to the evidence produced by the previously deployed technique. Regarding the information contained in an evidence instance, it can therefore be concluded that evidence produced by the alternative technique is at least as critical as the evidence produced by the previous technique. However, the alternative technique may produce evidence having additional information which increases the associated risk of unauthorized disclosure or alteration. Furthermore, the alternative measurement technique may possess some operational characteristics which increase operational risks as well as costs which should be considered when selecting a deployment variant.

EXAMPLE

Recall the example SQO *Every communication channel between the cloud service and a client using HTTP over an insecure network is secured using strong TLS cipher suites*. Let's assume that the cloud service under audit has changed in the following way: As result of increased security needs of the cloud provider, previously publicly reachable endpoints are now confined to only a few whitelisted hosts. Therefore, the non-invasive measurement technique which checked the supported TLS cipher suites by connecting to the endpoints is not feasible anymore. However, the cloud provider has exposed an existing Audit API which centrally exposes information about supported TLS suites of any of his service endpoints to authorized parties. Therefore, an alternative technique may call the Audit API to produce evidence and measurement results which are semantically similar to those results produced by the previous technique.

4 EXAMPLE APPLICATION LEVEL INTEGRATION

This section describes an example integration applying the process introduced in the previous chapter. The goal of this section is to show how the tool chain described in Section 2 can be integrated with SaaS applications on the application level. Application level integration allows to produce application level evidence and measurement results. This, in turn, permits to evaluate control objectives on the application level.

In the following section, we describe a dedicated EU-SEC Continuous Audit API which is developed in context of the Fabasoft Cloud, a SaaS application. Thereafter, Section 4.2 describes how the integration process introduced in Section 3.2 is applied to Fabasoft Cloud. Since this example integration is actually implemented as part of the continuous audit pilot in Working Package 5, Section 4.2 can only be completed once integration process as part of the pilot preparation (Task 5.1) is completed. Therefore, Section 4.2 is added later as part of Deliverable 3.5 (Deliverable 3.5 updates Deliverable 3.4).

4.1 EXAMPLE SAAS APPLICATION: FABASOFT CLOUD

To enable the tool chain to continuously audit cloud services on the application level, a measurement techniques' implementation such as Clouditor needs to be able to access a given API. In the following, we describe the development of a dedicated EU-SEC Continuous Audit API (EU-SEC CA API) with web services. The design of this API is driven by the requirements defined in Task 5.1 of Working Package 5 which is responsible for preparing the contiguous auditing pilot. The goal of this API is to be as agnostic as possible by basing its design on industrial standards. However, at certain points (environment, unique identifier structure, example calls) the EU-SEC CA API becomes application-specific.

4.1.1 ENVIRONMENT

Fabasoft provides two environments, one for development/testing and one for production usage. The environment for development/testing is the Fabasoft VDE (Virtual Development Environment). The production environment is the Fabasoft Cloud.

Fabasoft currently operates three data locations (governance regions). Each data location is addressed by a specific URL, one for Austria, one for Germany and one for Switzerland. The physical locations for these data locations are documented in the "Performance Characteristics Data Centers".[9]

The user accounts for customers of the Fabasoft Cloud are entirely managed by the customer, either by registering an account via https://www.fabasoft.com/register, by ordering a dedicated tenant via cloudsales@fabasoft.com or by invitation of an existing user in the Fabasoft Cloud.

4.1.2 EU-SEC CA API WEB SERVICES

The Fabasoft VDE/Fabasoft Cloud provides access for continuous auditing by standard protocols and by new, dedicated EU-SEC CA API web service calls, developed in this project.

The following base URLs must be used to access information in the Fabasoft VDE/Fabasoft Cloud:

- Fabasoft VDE: `https://vde.fabasoft.com/dev4/vm114/folio`
    - For testing & development purposes, e.g., Pilot 2 in Working Package 5
- Fabasoft Cloud
    - Data location „Austria": `https://at.cloud.fabasoft.com/folio`
    - Data location "Germany": `https://de.cloud.fabasoft.com/folio`
    - Data location "Switzerland": `https://ch.cloud.fabasoft.com/folio`

These base URLs are valid for both standard protocols and for dedicated EU-SEC CA API web service calls.

ACCESSING OBJECTS IN THE REPOSITORY

Each object in the Fabasoft VDE/Fabasoft Cloud is identified by a unique identifier. The identifier has the format "*COO.a.b.c.d*".

The Fabasoft VDE/Fabasoft Cloud provides two standard protocols to access all objects in the repository:

- CMIS (Content Management Interoperability Services)[10]
- WebDAV (Web-based Distributed Authoring and Versioning)[11]

---

[9] see https://www.fabasoft.com/data-center

[10] see https://en.wikipedia.org/wiki/Content_Management_Interoperability_Services, or http://docs.oasis-open.org/cmis/CMIS/v1.1/CMIS-v1.1.html

[11] see https://en.wikipedia.org/wiki/WebDAV or, https://tools.ietf.org/html/rfc4918

The Fabasoft VDE/Fabasoft Cloud provides the following entry points for the two standard protocols:

- CMIS

  `<baseurl>/cmis`

  e. g. `https://vde.fabasoft.com/dev4/vm114/folio/cmis`

- WebDAV

  `<baseurl>/webdav`

  e. g. `https://vde.fabasoft.com/dev4/vm114/folio/webdav`

- *CMIS SAMPLE*

The curl command line

```
curl -X GET -ukimble0001:PASSWORD "https://vde.fabasoft.com/dev4/vm114/folio/cmis"
```

will provide the following XML data – the highlighted line will provide the URL to the children of the root element of user kimble0001:[12]

```
<?xml version="1.0" encoding="UTF-8"?>

[…]

<title>FscDucx</title>
<app:collection href="https://vde.fabasoft.com/dev4/vm114/folio/cmis/COO.200.200.1.1975/COO.200.200.1.1975/children">
<title type="text">Root Collection</title>
<cmisra:collectionType>root</cmisra:collectionType>
</app:collection>

[…]

<cmisra:collectionType>templates</cmisra:collectionType>
</app:collection>
<cmisra:repositoryInfo>
<cmis:repositoryId>COO.200.200.1.1975</cmis:repositoryId>
<cmis:repositoryName>FscDucx</cmis:repositoryName>
<cmis:repositoryDescription></cmis:repositoryDescription>

[…]

</cmisra:uritemplate>
</app:workspace>
</app:service>
```

The curl command line

```
curl -X GET -ukimble0001:PASSWORD
"https://vde.fabasoft.com/dev4/vm114/folio/cmis/COO.200.200.1.1975/COO.200.200.1.1975/children"
```

will provide access to the children of the root element of user kimble0001 (with id `COO.200.200.1.1975`) and so on.

---

[12] More information about the Fabasoft CMIS implementation can be found here: https://help.folio.fabasoft.com/doc/Fabasoft-Integration-for-CMIS/index.htm

- *WEBDAV SAMPLE*

The curl command line

```
curl -X PROPFIND -H "Depth: 1" -ukimble0001:PASSWORD "https://vde.fabasoft.com/dev4/vm114/folio/webdav"
```

will provide the following XML data – the highlighted line will provide the id of the root element of user kimble0001:[13]

```
<?xml version="1.0" encoding="utf-8"?>
<D:multistatus xmlns:D="DAV:" xmlns:fsc="http://schemas.fabasoft.com/swc/">

[…]

<fsc:COOSYSTEM_1_1_objaddress
xmlns:fsc="http://schemas.fabasoft.com/swc/">COO.200.200.1.1975</fsc:COOSYSTEM_1_1_objaddress>

[…]

<D:status>HTTP/1.1 200 OK</D:status>
</D:propstat>
</D:response>
</D:multistatus>
```

AUTHENTICATION OF WEB SERVICES IN THE FABASOFT VDE AND CLOUD

Web Services to the Fabasoft VDE are authenticated via Basic Authentication, so the https requests of the web services must contain the basic authentication credentials (username and password).

Web Services to the Fabasoft Cloud are authenticated via Basic Authentication, but the password provided is a "Password for Application" configured in the account menu of the user, that wants to allow web service access.[14]

EU-SEC CA API WEB SERVICE CALLS

Fabasoft provides two calling conventions for accessing the EU-SEC CA API Web Services:

- SOAP
- JSON

The following calling conventions are used:

[13] More information about the Fabasoft WebDAV implementation can be found here: https://help.folio.fabasoft.com/index.php?topic=doc/Fabasoft-Integration-for-WebDAV/index.htm

[14] See https://help.cloud.fabasoft.com/index.php?topic=doc/User-Help-Fabasoft-Cloud-eng/account-menu.htm#access-for-applications for more information.

- SOAP:

  `<baseurl>/fscdav/wsdl?WEBSVC=EUSECCAAPI_111_100_WebService`
  e. g. `https://vde.fabasoft.com/dev4/vm114/folio/fscdav/wsdl?WEBSVC=EUSECCAAPI_111_100_WebService`

- JSON:

  `<baseurl>/wsjson/EUSECCAAPI_111_100_WebService/<webservicemethod>`
  e. g. `https://vde.fabasoft.com/dev4/vm114/folio/wsjson/EUSECCAAPI_111_100_WebService/CheckAccess`

In parameters are passed as a JSON object, a JSON object with the out parameters is returned.

4.1.3 EXAMPLES OF EU-SEC CA API REFERENCES

The example EU-SEC CA API consists of the following three endpoints:

- {hostname}/ca_api/datalocation/
- {hostname}/ca_api/encryption/
- {hostname}/ca_api/identityfederation/

The supported operations of the above endpoints can be found in Appendix A.

4.2 EXAMPLE INTEGRATION PROCESS

In this section, we describe how the integration process introduced in Section 4.2 is applied to Fabasoft Cloud, taking into account the EU-SEC CA API presented in the previous section. Since this example integration is also part of the continuous audit pilot in Working Package 5, this section can only be completed once integration process as part of the pilot preparation (Task 5.1) is completed. Therefore, this section is added later as part of Deliverable 3.5 (Deliverable 3.5 updates Deliverable 3.4).

5 EXAMPLE PLATFORM LEVEL INTEGRATION

This section describes another example integration applying the process introduced in Chapter 3.2. The goal of this section is to show how tool chain described in Section 2 can be integrated with IaaS on the platform level (i.e., integration with IaaS control plane). Platform level integration allows to produce platform level evidence and measurement results. This, in turn, permits to evaluate control objectives on the platform level.

In the following section, we describe a selection of IaaS provided Amazon Web Services (AWS). Thereafter, Section 5.2 describes how the integration process introduced in Section 3.2 is applied to AWS. Since this example integration is implemented as part of the continuous audit pilot in Working Package 5, Section 5.2 can only be filled once the integration process as part of the pilot preparation (Task 5.1) is finalized. Therefore, Section 5.2 is added later as part of Deliverable 3.5 (Deliverable 3.5 updates 3.4).

5.1 EXAMPLE IAAS: SELECTED AMAZON WEB SERVICES

5.1.1 ENVIRONMENT

Amazon Web Services (AWS) is the leading IaaS provider as of 2017 (3). The AWS Global Infrastructure[15] currently consists of 18 regions.

AWS follows the so-called shared responsibility model[16] which denotes that the responsibility to operate a cloud service secure is shared between the customer und AWS as a cloud provider: While AWS makes sure that its services are not vulnerable to attacks, customer have to configure AWS services which they use in a secure manner. This means that AWS takes no responsibility for, e.g., incorrectly configured customer security groups or vulnerable applications the customer may choose to deploy.

In this context, platform-level (or control plane) integration in the case of AWS delineates the integration of continuous auditing tool chain with an AWS customer. It does not mean, however, that the tool chain integrates with the underlying cloud infrastructure directly maintained by AWS.

---

[15] https://aws.amazon.com/about-aws/global-infrastructure/

[16] https://aws.amazon.com/compliance/shared-responsibility-model/

The current service portfolio of AWS consists of more than 100 services. With regard to the continuous auditing pilot of Working Package 5, the following four services are considered herein:

- Amazon Elastic Compute Cloud (EC2): Computing resource service
- Amazon Elastic Block Storage (EBS): Volumes for EC2 instances
- Amazon Simple Storage Service (S3): Object storage
- Amazon Rational Database Service (RDS): Managed rational database service supporting, e.g., MySQL
- AWS Key Management Service (KMS): Encryption and management of cryptographic keys

5.1.2 AWS APIS

Configuration information about EC2, EBS, S3, RDS and KMS required to determine whether control objectives are met on the platform level can be retrieved using the AWS API of the respective service. APIs are supplied as part of AWS SDKs which are available for multiple languages.[17]

5.1.3 EXAMPLE TEST-BASED MEASUREMENTS

Different to the example integration on application level where Fabasoft provided a dedicated Audit API to support control objective checks, AWS does not (yet) offer such an API on the platform level. Therefore, we have to draw on the AWS APIs to design the test-based measurement techniques outlined hereafter. These example techniques are selected based on the identified requirement provided by Task 5.1 of Working Package 5.

- Location of S3 objects: Determine location of data stored in S3 buckets.
- Encryption status of objects stored in S3 buckets: Determine if all objects stored in S3 are encrypted.
- Default encryption of object storage (bucket level): Determine if default encryption for an S3 bucket is enabled.
- S3 Encryption policy (bucket level): Determine if any S3 Bucket has an encryption policy
- Encryption of EBS volumes: Determine if all EBS volume are encrypted.
- Encryption status of databases provided by RDS: Determines all DB instances are encrypted.

---

[17] https://aws.amazon.com/tools/

- Origin of KMS keys: Determine if the KMS keys have the correct origin (expected: 'external')
- Key rotation of KMS keys: KMS keys have key rotation enabled (only applicable to non-external keys)

5.2 EXAMPLE INTEGRATION PROCESS

This section describes how the integration process introduced in Section 4.2 is applied to AWS, considering the example test-based measurement techniques presented in the previous section. This example integration is also part of the continuous audit pilot in Working Package 5 and therefore this section will be completed once integration process as part of the pilot preparation (Task 5.1) is completed. As a result, this section will be added later as part of Deliverable 3.5 (Deliverable 3.5 updates Deliverable 3.4).

6 EVALUATION OF CONTINUOUS TEST-BASED MEASUREMENT TECHNIQUES

As pointed out in the Introduction of this document, erroneous test results can decrease customers' trust in test results and can lead to providers disputing results of a continuous test-based security audits. In order to address this challenge, this chapter introduces a method how to experimentally evaluate the accuracy and precision of continuous test-based measurement techniques. This method allows to compare alternative test-based measurement techniques as well as compare alternative configurations of test-based techniques. Furthermore, it permits to infer general conclusions about the accuracy of a specific test-based measurement technique. Parts of the contents of this chapter have been published in (4), (5) and (6).

The next section introduces four universal metrics which can be used with any test-based measurement technique and, on this basis, defines the terms accuracy and precision in the context of such test-based techniques. Thereafter, Section 6.2 provides a high-level overview of how the method works and Section 6.3 describes how to violate of cloud service properties leading to dissatisfaction of SLOs or SQOs and thus non-compliance of the service with a certificate's controls. Then Section 6.4 introduces accuracy and precision measures applicable to any test-based measurement technique, including the inference of conclusions about the general accuracy of a test-based technique. Finally, Section 6.5 presents experimental results of applying our method to evaluate and compare exemplary continuous test-based measurement techniques which aim to support certification of controls related to property secure communication configuration.

6.1 BACKGROUND

In this section, first four universal test metrics are presented which can be used with any test-based measurement technique which strictly follows the building blocks described in Section 4.1 of Deliverable 3.2. Thereafter, Section 6.1.2 introduces basic measuring as well as statistical terminology and concepts which are required for experimental evaluation.

6.1.1 UNIVERSAL METRICS FOR TEST-BASED MEASUREMENT TECHNIQUES

Test-based measurement techniques seek to automatically and repeatedly produce measurement results which allow to check if a cloud service satisfies a set of objectives (i.e., SLOs and SQOs) over time. Recall that *metrics* take as input *evidence* provided by test-based techniques and output *measurement results*. Measurement results, in turn, are used to reason about SLOs and SQOs. Continuous test-based measurement therefore implies that a sequence of instances of evidence have to be interpreted by suitable metrics in order to produce measurement results which, in turn, allow to reason about defined objectives over a period of time.

Recall that in Section 4.1 of Deliverable 3.2, the building blocks of test-based measurement techniques to continuously produce measurement results to be used for security audits were presented. *Test cases* form the primitive of each continuous test which use test oracles to determine the outcome of a test case, that is, whether a test cases passes or fails. Further, *test suites* combine test cases where each suite contains at least one test case. A test suite either passes or fails, it passes if all contained test cases pass.

Note that the definition of *metric* used here refines the one provided by Deliverable 1.4: We describe a metric as a function $M: R \rightarrow U$ which takes as input results of test suite runs R and outputs measurement results U. A metric can be computed based on 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.

Any test metric used by a test-based measurement technique which strictly follows the building blocks defined in Section 4.1 of Deliverable 3.2 can therefore make use of the following two characteristics: First, a single test suite run (i.e., a single execution of a test suite as part of a continuous test) either passes or fails. As a consequence, and second, a single test suite run passes or fails *at some point in time*. Based on these two key characteristics, four test metrics functions are proposed hereafter which are universally applicable to any type of evidence.

BASIC-RESULT-COUNTER (BRC)

A basic test result br tells us if a test failed (f) or passed (p), i.e., $br \in \{f, p\}$. The Basic-Result-Counter (brC) metric takes any instance of *br* as input and counts the number of times a test failed (brC^F) or passed (brC^P).

As Figure 6-2 shows, a basic test result is only returned after the execution of a test suite run completed (tsr_i^s). This metric can be used to assess statements only requiring to evaluate if and how often a continuous test failed or passed. Consider, as an example application, determining if and how often security groups assigned to a newly started virtual machine unexpectedly allow that these machines are publicly accessible through other than whitelisted ports.

FAILED-PASSED-SEQUENCE-COUNTER (FPSC)

A continuous test repeatedly produces basic test results. A *failed-passed-sequence* (fps) is a special sequence of basic test results: As Figure 6-1 shows, a fps starts with a failed test at t_i given that the previous test at t_{i-1} passed. An fps ends with next occurrence of a passed test.

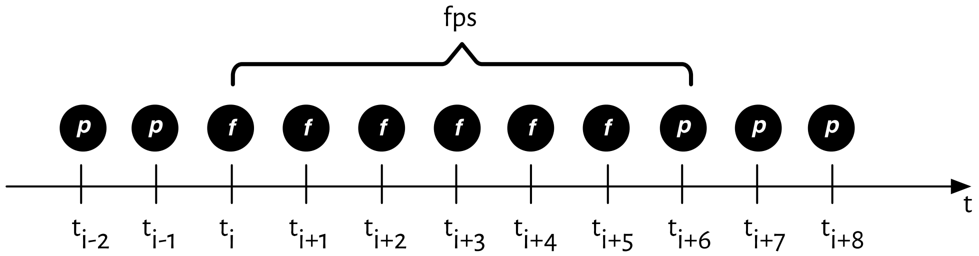


Figure 6-1 Exemplary failed-passed-sequence (fps) based on basic test results (br)

For example, consider having observed the following sequence of basic test results produced by a continuous test: When attempting to connect to a VM for eleven times in a row, the first two times the login were successful (p). However, for the next six times, the login fails (f) and for the remaining three times, the test succeeds again. The example fps is the sequence $\text{f}ps_{ssh}^{11} = \langle f, f, f, f, f, f, p \rangle$.

The Fail-Pass-Sequence-Counter ($fpsC$) metric uses this definition of fps. $fpsC$ counts the number of occurrences of fps which are observed within a sequence of basic test results $S_{br} = \langle br_1, br_2, \ldots, br_i \rangle$ produced during a continuous test. Consider, as an example, Figure 6-1 which shows the following sequence of basic test results $\hat{S}_{br} = \langle p, p, f, f, f, f, f, p, p, p \rangle$. Sequence $\hat{S}_{br}$ contains exactly one fps, i.e., $fpsC(\hat{S}_{br}) = 1$.

FAILED-PASSED-SEQUENCE-DURATION (FPSD)

The Fail-Passed-Sequence-Duration ($fpsD$) metric draws on the definition of a failed-passed-sequence (fps). $fpsD$ takes a fps as input and measures the time between the first failed test of an fps and its last basic test result which passes by definition. This test metric allows to reason about properties over individual periods of time, thus it can be used to evaluate statements which contain time constraints. Consider, for example, a control implementation derived from, e.g., *RB-21: Handling of vulnerabilities, malfunctions and errors – check of open vulnerabilities* of BSI C5 (7) that an incorrectly configured and thus insecure webserver's TLS setup of a SaaS application is fixed within a certain amount of time, e.g., eight hours.

The definition of $fpsD$ has a subtle detail: Recall that $fpsD$ aims to measure the time difference between the first and the last test of a *failed-passed-sequence*, that is,

$$fps = \langle \boldsymbol{f_i}, f_{i+1}, f_{i+2}, \dots, \boldsymbol{p_{i+j}} \rangle.$$

It is important to note at this point is that the first failed test f_i as well as the next passed test p_{i+j} each have a duration themselves. This means that both tests take some time to complete and return a basic test result. As a consequence, we have to select whether a $fpsD$ starts at the start time or the end time of the first test f_i. Further, we have to decide whether a $fpsD$ ends at the start time or the end time of the last test p_{i+j}.

In order to properly define the limits of a $fpsD$, we have to first shed light on different options which may affect our metric. For example: Figure 6-2 illustrates the definition of $fpsD$ which uses the start time tsr_i^s of the first failed test tsr_i and the end time tsr_{i+j}^e of the next passed test tsr_{i+j}. Note that duration of the first failed test is d_i and duration of the last passed test is d_{i+j}.

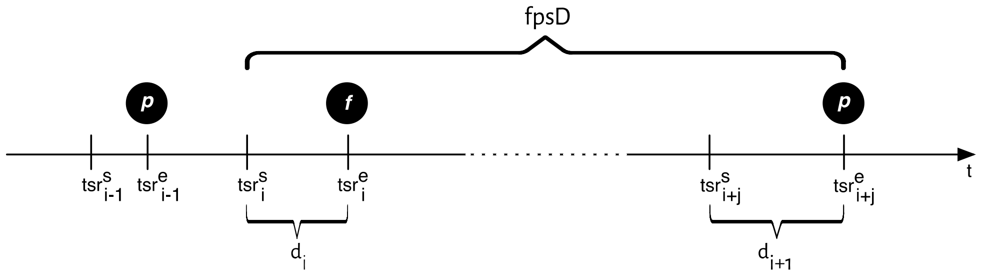


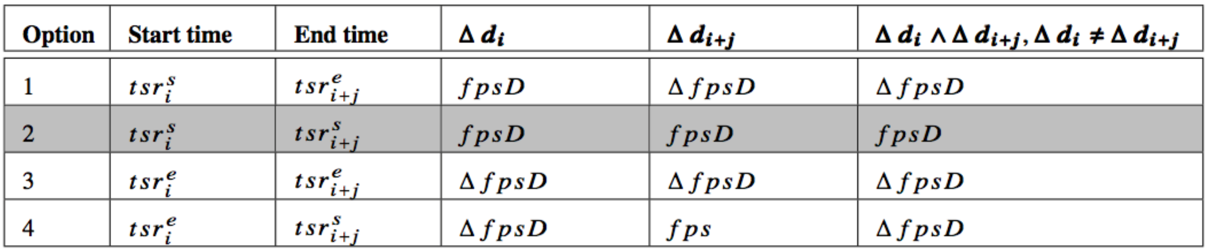
Figure 6-2 Example definition for universal test metric $fpsD$

It is obvious that the example definition of $fpsD$ shown in Figure 6-2 has a downside: The more time it takes the last test tsr_{i+j} to complete, the higher the proportion of d_{i+j} within the $fpsD$. Therefore, choosing tsr_i^s and tsr_{i+j}^e as bounds for $fpsD$ makes $fpsD$ dependent on the duration of tsr_{i+j}. For scenarios requiring high accuracy of $fpsD$, e.g., to evaluate statements defining narrow time constraints, this dependency can make the metric $fpsD$ unsuited.

As already pointed out in the introduction of this section, the metric $fpsD$ ought to be applicable to any continuous test. This means that a definition of $fpsD$ has to avoid dependencies of the duration of a specific last test tsr_{i+j}. In order to derive a definition of $fpsD$

least dependent on test suite runs' duration, we have to analyze how variations in the duration of the first failed test (d_i) and the last passed test (d_{i+j}) impact on $fpsD$.

Figure 6-3 shows the four available options to define $fpsD$. Let us consider, for example, *Option 3*: Here, the end of the first failed test (tsr_i^e) is used as start of the $fpsD$ while the end of the next passing test (tsr_{i+j}^e) serves as end of the $fpsD$. When selecting this definition, variations of either the duration of the first test (Δd_i) as well as the last test (Δd_{i+j}) will impact on the $fpsD$, i.e., result in $\Delta fpsD$. Also, variations of both tests ($\Delta d_i \wedge \Delta d_{i+j}$) also change $fpsD$, i.e., $\Delta fpsD$. Note that there exists a corner case where duration variations of both tests cancel each other out, that is, if $\Delta d_i = d_{i+j}$, then $fpsD$ remains unaffected.



Option	Start time	End time	Δd_i	Δd_{i+j}	$\Delta d_i \wedge \Delta d_{i+j}, \Delta d_i \neq \Delta d_{i+j}$
1	tsr_i^s	tsr_{i+j}^e	$fpsD$	$\Delta fpsD$	$\Delta fpsD$
2	tsr_i^s	tsr_{i+j}^s	$fpsD$	$fpsD$	$fpsD$
3	tsr_i^e	tsr_{i+j}^e	$\Delta fpsD$	$\Delta fpsD$	$\Delta fpsD$
4	tsr_i^e	tsr_{i+j}^s	$\Delta fpsD$	fps	$\Delta fpsD$

Figure 6-3 Available options to define Fail-Pass-Sequence-Duration ($fpsD$) if $|fps| > 2$

When inspecting Figure 2-1, it is obvious that *Option 2* is the only definition of $fpsD$ unaffected by variations of duration of the first and the last test suite run. Therefore, we define the *start* of a $fpsD$ to be the start time of the first failed test (i.e., tsr_i^s) while the *end* of a $fpsD$ is the start time of the next passed test (i.e., tsr_{i+j}^s).

Note that the reasoning shown in Figure 6-3 is only true if the *failed-passed-sequences* contains more than two basic test results, that is, $|fps| > 2$. In case a fps only containing two elements, i.e., $fps = \langle f_i, p_{i+1} \rangle$, then variations of the duration of the failing test f_i will lead to changes of $fpsD$. Furthermore, if $|fps| = 2$, then the duration of the $fpsD$ will be at least as long as it takes the failing test to complete. Consequently, the time it takes to complete the first failing test also defines the lower bound on how accurately we can reason about statement containing time constraints.

CUMULATIVE-FAILED-PASSED-SEQUENCE-DURATION (CFPSD)

This metric builds on the *failed-passed-sequence-duration* ($fpsD$) presented in the previous paragraph. The input to test metric $cfpsD$ is a sequence $\hat{S}_{fpsD}$ consisting of any $fpsD$ observed during a continuous test, and, on this basis, $cfpsD$ outputs their accumulated value.

The metric $cfpsD$ allows us to reason about cloud service properties within a predefined period of time. Similar to the metric $fpsD$, we can leverage $cfpsD$ to evaluate statements containing time constraints. Different to $fpsD$, however, $cfpsD$ permits to evaluate statements whose time constraints refer to multiple property violation events observed within a particular period of time. As an example, consider a service level agreement which defines that the total yearly downtime of a cloud service must not surpass five minutes (Note that compliance with SLAs is required by various controls, e.g., RB-02 Capacity management – monitoring of the Cloud Computing Compliance Controls Catalogue (BSI C5) (7)). During the period of a year, the cloud service exhibits multiple, timely separated downtime events which are detected by a suitable continuous test. The metric $fpsD$ can be used to evaluate statements which contain a single downtime event to, e.g., not last longer than 60 seconds. In contrast, $cfpsD$ takes a period of time into account, e.g., a year, and summarizes over any $fpsD$ observed to evaluate statements that refer to all downtime events during the entire period.

6.1.2 ACCURACY AND PRECISION

In the previous section, four universal test metrics for test-based measurement techniques have been introduced which allow us to evaluate SLOs and SQOs defined for cloud services. The question at this point is: What errors do these measurements results possess and how do these errors affect our conclusion about whether a cloud service satisfies a SLO or SQO.

In this section, it is will defined what *accuracy* and *precision* mean in the context of measurement results produced by the four universal test metrics. To that end, we draw on standard measurement theory and statistical methods used within various fields of experimental science. The basic definitions of concepts such as *accuracy* and *precision* used within this section follow (8), (9) and (10). Furthermore, statistical methods leveraged within this section are comprehensively covered in the literature, e.g., (11) (12) (13).

ACCURACY

The *accuracy* of the measurement describes whether the measured value agrees with the accepted value. This *accepted* or *true* value can be provided by previous observations or theoretical calculations. The concept of accuracy thus only applies if experimental data is analyzed with the goal to compare the experimental results with known values.

Recall the four test metrics *brC*, *fpsC*, *fpsD* and *cpfsD* which have been introduced in Section 6.1.1. The accuracy of measurement results produced by these test metrics are outlined hereafter:

- *Basic-result-Counter (brC):* This metrics counts the number of passed and failed tests. A basic test result is accurate if it indicates that a control is not satisfied by the cloud services at a time where the cloud service indeed does not comply with the control. Also, a basic test result is accurate if it indicates satisfaction of a control by a cloud service at a time where the service indeed complies with the control.
- *Failed-Passed-Sequence-Counter (fpsC):* This metrics counts the number of observed failed-passed-sequences (*fps*). A *fps* is accurate if the cloud service actually does not comply with a control during the time indicated by the *fps*.
- *Failed-Passed-Sequence-Duration (fpsD):* This metric describes the time elapsed between the first failed test and the last passed test of a *fps*. A measurement result produced by *fpsD* is accurate if it agrees with the actual duration of temporary non-compliance of a cloud service.
- *Cumulative-Failed-Passed-Sequence-Duration* (*cfpsD*): This metric describes the accumulated time during which a control is not satisfied. A measurement result produced by *cfpsD* is accurate if it matches the acutual duration of the temporary non-compliance of cloud service within a specified interval.

The reason why measured values may not agree with accepted values *are systematic errors*. These errors may result from, e.g., erroneous implementation and configuration of the measuring device. Identifying the causes of systematic errors is usually non-trivial where, in the case a test-based measurement technique, this measuring device consists of any component used to implement the test-based measurement technique, that is, any component implementing the framework to design continuous test described in Chapter 4 of Deliverable 3.2.

Systematic errors of measurement results vary depending on the test metric. In Section 6.4, accuracy measures for each of the four universal test metrics will be explored which allows to quantify the disagreement between measured values and true values.

Furthermore, as will be detailed in Section 6.3, true values are established through intentionally manipulating cloud services to not satisfy a SLO or SQO which measurement results produced by the universal test metrics aim to check. Thus, we know the true values and can compare them with the measured ones provided by the evidence production technique under evaluation, thereby providing us with the accuracy of the technique. However, the remaining problem is that the systematic error measurement results may exhibit can vary due to *random errors*. This brings us to the concept of *precision* which is explained in the following section.

PRECISION

Precision refers to the closeness of agreement between successively measured values conducted under identical conditions (9). When neglecting systematic errors, then those repeatedly executed measurements provide a range of values spreading about the true value. The reason for this spread are *random errors* which are caused by unknown and unforeseeable changes in the experiment, e.g., fluctuation in the network delay to due electronic noise. The smaller the random errors, the smaller the range of values, and thus the more precise the measurement (8). Hence, the level of precision of experimental measurements is determined by random errors.

- *Arithmetic mean*: Assume having observed some repeated measurements $X = \langle x_1, x_2, \dots, x_n \rangle$ only having random errors. The question now is: What is true value of these measurements? In statistical terms, the answer is to use the values of sample distribution X to estimate the expected value μ of the parent distribution Y. The best estimate for μ to be derived from these measurements is the arithmetic mean. Using the values of X, we compute the sample mean

  $$\bar{x} = \frac{1}{n}\sum_{i=0}^{n} x_i$$

  serving as our estimate of μ. Averaging follows the intuition that random errors are equally likely to be above as well as below the true value. Thus, averaging evenly divide the random error among all observations.

  A special case arises if the values of X and Y can only assume one of two values, for example, 0 or 1. In this case, computing the arithmetic mean give us the fraction of values with 1's of X. This is referred to as the sample *proportion* $\bar{p}$ which serves as an estimate of the population proportion p.

  At this point, it is important to note that the assumption of our measurements in X only having random errors is rather theoretical. In a real experiment, each $x \in X$ will possess random errors and systematic errors. Therefore, $\bar{x}$ or $\bar{p}$ are *not* estimates for their true value, they provide estimates for their true values *plus* their systematic errors.

  Estimating the population mean μ and population proportion p based on $\bar{x}$ and $\bar{p}$ works because of the laws of large numbers: The *weak law of large numbers* states that if the number of samples n generated from the distribution Y goes to infinity, then the probability of making a random error larger then ϵ goes to zero:

  $$\lim_{n \to \infty} P(|\bar{x}_n - \mu|) > \epsilon = 0.$$

Furthermore, the *strong law of large numbers* states that the probability of the sample mean $\bar{x}_n$ converging to the expected value is 1:

$$P(\lim_{n\to\infty} |\bar{x}_n - \mu| = 0) = 1.$$

Both laws of large numbers suggest that provided a sufficiently large number of samples, – i.e., take a sufficient large number of measurements –we can produce an estimate $\bar{x}$ with a random error $\epsilon = |\bar{x}_n - \mu|$ which can be as small as we desire. Put differently: Given a sufficiently large number of measurements, the estimate converges to the true value plus systematic error. Yet neither law tells us how many measurements have to be conducted to reduce ϵ below a particular threshold.

- *Standard deviation:* The sample mean $\bar{x}$ estimates the true value plus systematic errors. However, it does not provide us with any information on the range of measured values. To describe the width of the sample distribution X, we can use the standard deviation

  $$sd = \sqrt{\frac{1}{|X|}((x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \cdots + (x_i - \bar{x})^2)}.$$

  The standard deviation considers any values of X and provides the average distance of a measurement value to the mean. If we observe another measurement and want to know if it is a common or exceptional value, then we can make use of sd. First, we standardize the observed value x by computing *z-scores*:

  $$z = \frac{(x - \bar{x})}{sd}.$$

  Whether a z value is low or high depends on the distribution of X: In case of a normal distribution, 99% of the values lie within z-scores of [-3,3] where any value outside this range may be considered exceptional.

  The sd has one important disadvantage: Adding more measurement values to X increases the precision with which we can estimate the population mean μ since it decreases the random error. Yet when conducting more measurements, the standard deviation of X remains relatively stable. This means that the standard deviation is *not* a good measure to describe the error of the sample mean, that is, the closeness of the sample mean to the population mean.

- *Standard error:* Having estimated the population mean μ with $\bar{x}$, the standard error se is the suitable choice when intending to describe the precision of $\bar{x}$. The se is the standard deviation of the so-called *sampling distribution*. Note that we have already seen two distributions, that is, the parent distribution Y whose expected value we aim to estimate using sample distribution X which contains the samples drawn from Y. The *sampling distribution* is a theoretical distribution which were to obtain if we draw all possible samples X from Y and compute a statistic, e.g., the mean of each of these samples. Naturally, in practice, this is usually impossible or not desired. The resulting distribution of all these samples means is the *sampling distribution of the mean*.

The calculation of the standard error depends on the statistic. The se for the sample mean $\bar{x}$ can be obtained as follows:

$$se_{\bar{x}} = \frac{sd_{\bar{x}}}{\sqrt{n}}.$$

It is obvious that an increasing standard deviation sd of the sample distribution X leads to a higher standard error. However, the standard error decreases if the number of samples in X, that is, n increases.

Further, the standard error for a sample proportion $\bar{p}$ is computed as follows:

$$se_{\bar{p}} = \sqrt{\bar{p} \times \frac{(1-\bar{p})}{n}}.$$

- *Confidence intervals:* Combining the notion of the standard error with the assumption that the sampling distribution approximately follows a normal distribution permits estimating the precision of the sample mean and the sample proportion by constructing confidence intervals for the sample mean and for the sample proportion. In contrast to point estimation like $\bar{x}$ and $\bar{p}$, *confidence intervals* are a special type of interval estimates which give a range of probable values of an unknown parent's distribution parameter.

  In order to construct a confidence interval, it is necessary to decide on a confidence level and then compute the desired statistic, e.g., sample mean $\bar{x}$, as well as the margin of error (E).

  - *Confidence level (CL):* The fraction of all possible samples expected to include the true parameter of the unknown parent distribution. Consider, as an example, all possible samples X are drawn from the distribution of Y and for each a 99% confidence interval for the sample mean is computed. In this case, 99% of the computed confidence intervals would include the population mean, i.e., the mean of the distribution of Y.

  - *Statistic:* The property of a sample which is used to estimate population parameter's value. In our case, we use the sample mean $\bar{x}$ and the sample proportion $\bar{p}$.
  - *Margin of error (E):* This margin defines the interval estimation by the the range above and below the sample statistic. The calculation of E depends on the standard error which, in turn, depends on the selected statistic. For the sample mean $\bar{x}$, the margin of error is
$$E_{\bar{x}} = t_{CL} \times se_{\bar{x}}.$$
t_{CL} is the value that separates the middle the area of the t-Distribution according to the selected confidence level CL, e.g., 95%, and the standard error of the mean $se_{\bar{x}}$.
For the sample proportion $\bar{p}$, the margin of error is
$$E_{\bar{p}} = z_{CL} \times se_{\bar{p}}.$$
z_{CL} is the z-value that separates the middle area of the standard normal distribution according to the chosen confidence level CL, e.g., 99%, and the standard error for the proportion $se_{\bar{p}}$.
- *Calibrating precision:* Recall that at the end of paragraph on the arithmetic mean, it was described that the laws of large numbers justify making a point estimate of a parent's distribution parameter, e.g., using the sample mean $\bar{x}_n$ to estimate the mean μ of the distribution of Y. Yet we do not know how close this estimate is to the true value (plus systematic error), that is, how large is the error ϵ for a given sample size n?

After introducing confidence intervals' construction for sample means and proportions, we can now leverage the following idea: The sample size n can be used as a parameter to determine the number of samples needed to achieve a desired margin of error $\hat{E}$, that is, the desired precision. To that end, $E_{\bar{p}}$ and $E_{\bar{x}}$ are solved for the sample size n which gives us

$$\hat{n}_{\bar{p}} = \frac{z_{CL} \times \bar{p} \times (1 - \bar{p})}{\hat{E}^2}$$

and

$$\hat{n}_{\bar{x}} = \frac{sd_{\bar{x}} \times t_{CL}^2}{\hat{E}^2}.$$

In practice, one apparent problem of solving these formulas is that they have to be solved prior to executing the experiment to evaluate a test-based measurement technique. This means that there may not exist any previously observed values to plug in for $\bar{p}$ and $sd_{\bar{x}}$. This would leave us with an educated guess of these values, otherwise we may use historical values previously observed.

6.2 OVERVIEW OF THE EVALUATION PROCESS

The accuracy and precision of measurement results produced by a specific test-based measurement technique depend on various factors, such as implementation of the test, test environment and usage of external tools. Without experimental evaluation, it is thus hard to make a statement on how well a test-based measurement technique works in detecting a control's (i.e., a SLO's or SQO's) satisfaction or violation.

The approach described hereafter treats a test-based measurement technique under evaluation as a black box. Therefore, no information about the internal composition and implementation of the technique is needed, e.g., if and which external tools are used. Only measurement results produced by the test-based measurement technique during an experiment are observed where the violations of the control, that is, violations of the SLOs or SQOs associated with the control are induced which the technique intends to validate. Put differently: Correct results as well as errors of the test-based technique under evaluation follow some unknown distributions. Samples from these unknown distributions are taken by running experiments where controls are intentionally violated. Based on these experiment results, conclusions about the accuracy of the test-based measurement technique are drawn.

Figure 6-4 provides a high-level overview of our method. As part of configuring a control violation sequence, duration of and time between each control violation event is randomized within some specified limits (Step 1). Then the test-based technique is configured according to the building blocks described in Deliverable T3.2, Section 4 (Step 2): Selecting test cases, setting test suites parameter and choosing a workflow. Thereafter, the control violation sequence and the test-based technique are started at the same time (Step 3). Then it is observed whether violation events are detected by the test-based measurement technique (Step 4). Provided the sample size is sufficiently large, i.e., enough measurement results have been produced (Step 4), the parameters of the unknown parent distribution are inferred, that is, we draw conclusions about the general accuracy of the test-based technique under evaluation (Step 5). These inferences are considered valid with regard to the test and control violation configuration parameters.

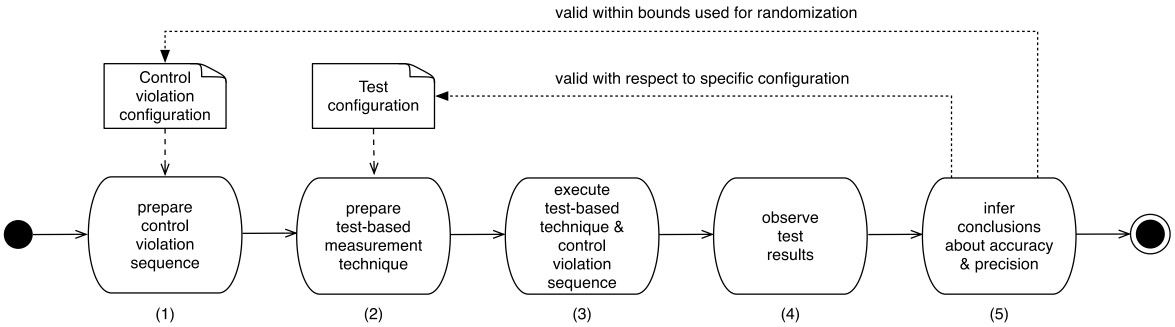

valid within bounds used for randomization
Control violation configuration
Test configuration
valid with respect to specific configuration
prepare control violation sequence
prepare test-based measurement technique
execute test-based technique & control violation sequence
observe test results
infer conclusions about accuracy & precision
(1)
(2)
(3)
(4)
(5)

Figure 6-4 Experimental evaluation of the accuracy and precision of test-based measurement techniques

6.3 SECURITY CONTROL VIOLATION

In this section, it is described how to violate controls of a cloud service which a test-based measurement technique aims to detect, that is, which the technique's evidence is expected to indicate. Thereby, the ground truth is established which allows to reason about correctness of evidence produced by specific test-based measurement technique.

6.3.1 CONTROL VIOLATION SEQUENCE

Recall that one of the key drivers for continuously testing cloud services is founded on the assumption that a cloud service's property is non-stationary, that is, may change over time where these changes can lead to control violations. This means that the properties of cloud service may comply with a control at some time while at other times, they do not.

In order to mock such non-stationary behavior of cloud services' properties, control violations have to continuously, i.e., repeatedly create control violation events (cve) over time. During a cve, a cloud service's properties are manipulated so that the service does not comply with the control, i.e., not to satisfy the SLOs and SQOs associated with the control. Between two successive cve, the cloud service's properties satisfy meet relevant SLOs and SQOs. This control violation sequence can be described as follows:

$V = \langle cve_1, cve_2, \dots, cve_i \rangle.$

As Figure 6-5 shows, each cve starts at cve^s and ends cve^e, thus having a duration of

$$cveD = cve^e - cve^s$$

where the service does not comply with the control. Furthermore, the time between two successive control violation events cve_{i-1} and cve_i is

$$cveW = cve_{i-1}^{e} - cve_{i}^{s}.$$

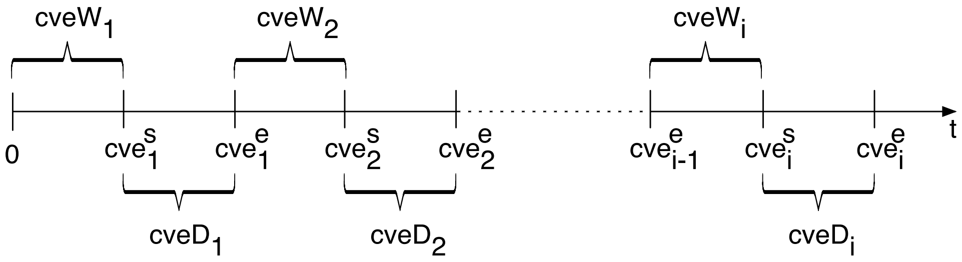


Figure 6-5 Sequence of control violation events cve

6.3.2 CONTROL VIOLATION DESIGN

The design of a control violation is driven by the specific control for whose validation the test-based measurement technique under evaluation aims to provides measurement results. Therefore, the question at this point is: Which properties of a cloud service have to be altered to violate a particular control?

It is important to note at this point that it is *not* the aim here to design control violations which are *complete*, that is, which manipulates a cloud service in *any* possible way such that a particular control is not satisfied. While such a complete control violation design would be helpful to evaluate the completeness of the test-based measurement technique, designing such a complete control violation faces similar challenges to deriving suitable test metrics from high-level, ambiguous SLO and SQO definition: To that end, interpreting what it means for a specific control to be satisfied or dissatisfied on the implementation level of a cloud service instance is needed. The difference to deriving test metrics is, however, that we were to design mechanisms intentionally manipulating a cloud service's properties to violate the control.

The goal of our evaluation is *correctness* of a test-based measurement technique, that is, the goal is to evaluate how accurate and precise the results produced by the test-based technique under evaluation are. Therefore, the test configuration of the continuous test under evaluation can serve as a starting point to derive the design of the control violation.

The control violation design process consists of two major steps:

1. *Inspect assert parameter:* The first step consists of inspecting the configuration of the test-based measurement technique under evaluation. Recall that a single test result of a test-based technique test is produced by executing a test suite which fails if any test case bound the test suite fail (see Deliverable 3.2, Chapter 4 for further detail). Therefore, the assert parameters which are used to configure the expected outcome of each test case are inspected. Based on the assert parameters and on their configured value, it can be determined which property of the cloud service has to be manipulated in order for these asserts to not be satisfied.
   Consider, as an example, that a test-based measurement technique probes a set of ports to check if the cloud service exposes sensitive interfaces. The assert parameters of the test definition will denote the ports which are considered sensitive, that is, should not be reachable. A control violation event may, e.g., manipulate the service's properties such that it exposes the blacklisted ports.
2. *Specify control violation events:* The second step consists of deciding on the lower ($cveW^L$) and upper ($cveW^R$) limit of the interval between two successive control violation events $cveW$. Furthermore, the lower ($cveD^L$) and upper ($cveD^R$) limit of the time during which a cloud service's property is manipulated to render it non-compliant have to be defined. The following section explains the purpose of randomizing duration of and interval between control violation events. Note that deciding on how many control violation events a control violation sequence should consist of is driven by the selected precision measures which are explained in detail in Section 6.4.

6.3.3 STANDARDIZING CONTROL VIOLATION EVENTS

Control violation sequences establish the ground truth against which specific test-based measurement techniques are evaluated. To infer conclusions about the general accuracy of a test-based measurement technique, ideally any possible sequence of any possible control violation event has to experimentally evaluated. Naturally, this is infeasible in practice and a sequence of control violation events V has to be selected which meets tolerable time and space constraints.

But how to select a sequence V which allows to draw conclusions about the general correctness of a test-based technique? The answer consists of two parts: At first, a control violation event needs to be standardized: For each cve we use to construct V, the duration of the control violation $cveD$ and the waiting time before start $cveW$ are selected randomly from intervals $[cveD^L, cveD^R]$ and $[cveW^L, cveW^R]$, respectively. Choosing these intervals' limits permits to configure control violations according to tolerable space and time limitations. Secondly, it needs to be decided how many cve, i.e., $|V|$ are required to infer conclusions about the general accuracy and precision of the test-based measurement technique test under evaluation. This depends on the statistical inference method which, in turn, depends on the precision measure. This is addressed for each precision measure in the following Section.

6.4 ACCURACY AND PRECISION MEASURES

This section describes models to estimate the accuracy and precision of test-based measurement techniques. Hereafter, these models are referred to as *accuracy measures* and *precision measures*. These measures are based on the universal test metrics brC, $fpsC$, $fpsD$, and $cfpsD$ introduced in Section 6.1.1.

In order to derive the accuracy and precision measures, each of the next four sections (6.4.1–6.4.4) follow these three steps:

1. *Evaluate measurement results:* The measurement results produced by a test-based measurement technique during a control violation sequence are used to evaluate to determine whether they are correct or erroneous. In the latter case, the type of observed error is specified which depends on the universal test metric used, e.g., a false negative basic test result incorrectly suggesting that a cloud services does not satisfy a control.
2. *Derive accuracy measures:* Using the evaluation of the measurement results as input, the accuracy measures then estimate if and how the measured values produced by test-based measurement techniques under evaluation deviate from the accepted, i.e., true values as established by control violation sequences.
3. *Derive precision measures:* Based on the evaluation measures, the precision measures estimate of and how the measured values spread about the accepted value.

6.4.1 BASIC-RESULT-COUNTER

This section describes how to estimate accuracy and precision of measurement results using the Basic-Result-Counter test metric (brC). To that end, the next section describes the evaluation of measurement results using different evaluation measures. Thereafter, it is detailed how to use these evaluation measures to compute accuracy and precision measures.

EVALUATION OF MEASUREMENT RESULTS

Hereafter, it is explained how to use the Basic-Result-Counter metric (brC) to evaluate a test-based measurement technique. To that end, we check whether measurement results correctly indicated absence or presence of a control violation event. Recall that brC^F and brC^T count failed br^F and passed test results br^T, respectively. Furthermore, each test tsr producing a basic test result br starts at tsr^s and ends at tsr^e, having a test duration of $tsrD$.

- *True negative basic test result counter* (brC^{TN}): A test produces a true negative result if the test fails at a time when a control is violated. As shown in Figure 6-6, a br^{TN} is produced if a failing test starts (tsr^s) after a control violation event starts (cve^s) and the test ends (tsr^e) before the event ends (cve^e):

$br^{TN} = cve^s \leq tsr^s \wedge tsr^e \leq cve^e$.

We count any the true negative test results observed during the control violation sequence. As a result, we obtain brC^{TN}.

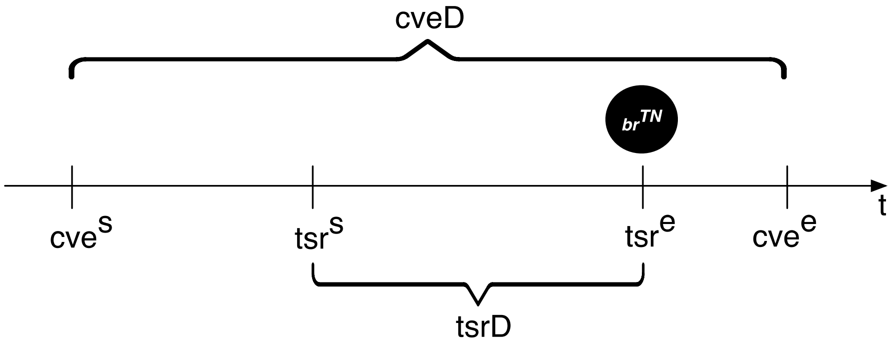


Figure 6-6 True negative basic test result (br^{TN})

- *True positive basic test result counter* (brC^{TP}): A true positive test result is produced if the test passes at a time when *no* control is violated. As shown in Figure 6-7, a passing test producing a true positive result starts after the previous control violation event ends and ends before the next control violation event starts:

$$br^{TP} = cve_i^e < tsr^s \wedge tsr^e < cve_{i+1}^s.$$

There are two special cases: First, a test which passes prior to any control violation event is a true positive. Therefore, any passing test which ends (tsr^e) before the first violation event starts (cve_1^s) is a true positive:

$$br^{TP} = tsr^e < cve_1^s.$$

Second, a test that passes after the last control violation even is a true positive test result. Thus any passing test which starts (tsr^s) after the last control violation event j ends (cve_j^e) is a true positive:

$$br^{TP} = cve_j^e < tsr^s.$$

Any true positive basic test result which is observed during a control violation sequence is counted using brC^{TP}.

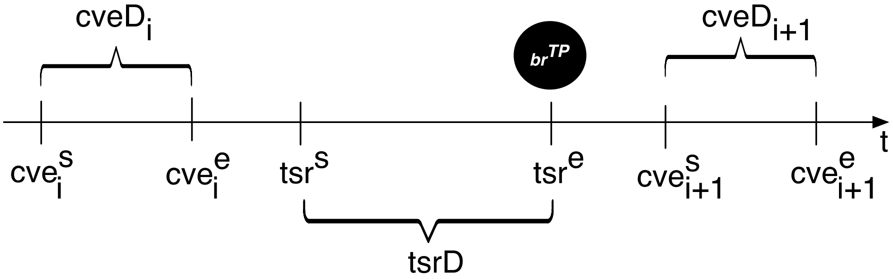


Figure 6-7 True positive basic test result (br^{TP})

- *False negative basic test result counter* (brC^{FN}): If a test fails at a time when *no* control is violated, then the test produces a false negative test result. When comparing Figure 6-7 and Figure 6-8, it becomes evident that the definition of a false negative test result is analogous to the definition of a true positive test result. The only difference being that the test result incorrectly fails:

$$br^{FN} = cve_i^e < tsr^s \wedge tsr^e < cve_{i+1}^s.$$

Furthermore, similar to true positive results, two special cases exist: First, a test that incorrectly fails prior to any control violation event is a false negative. Therefore, any failing test which ends (tsr^e) before the first violation event starts (cve_1^s) is a false negative:

$$br^{FN} = tsr^e < cve_1^s.$$

Second, a test that incorrectly fails after the last control violation event is a false negative test result. Therefore, any failing test which starts (tsr^s) after the last control violation event j ends (cve_j^e) is a false negative:

$$br^{FN} = cve_j^e < tsr^s.$$

Any false negative basic test result which are observed during a control violation sequence is counted using brC^{FN}.

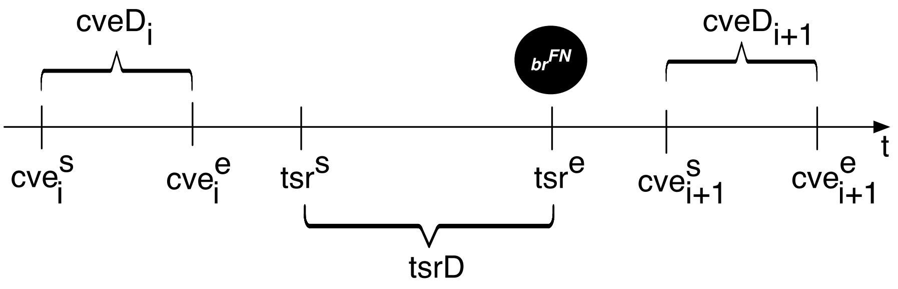


Figure 6-8 False negative basic test result (br^{FN})

- *False positive basic test result counter* (brC^{FP}): If a test passes at a time when a control is violated, then the incorrectly passing test produces a false positive result (br^{FP}). The definition of br^{FP} is similar to a true negative result (see Figure 6-6), only that the test incorrectly passes:

$$br^{FP} = cve^s \leq tsr^s \land tsr^e \leq cve^e.$$

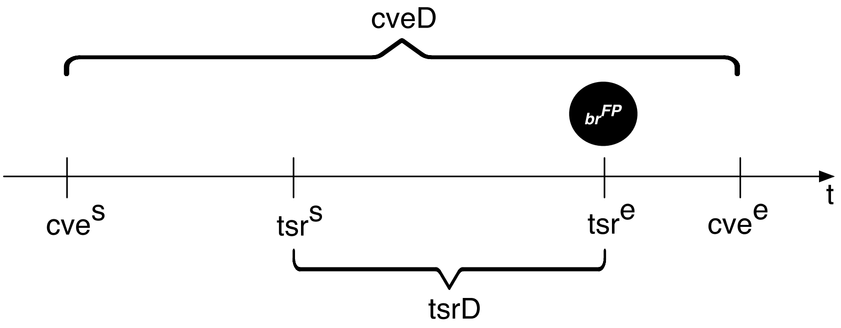


Figure 6-9: False positive basic test result (br^{FP})

Also, there is one special case: As shown in Figure 6-9, a passing test may cover one or more control violation events completely:

$$br^{FP} = cve_i^e < tsr^s \land tsr^s < cve_{i+1}^s \land cve_{i+j}^e < tsr^e \land tsr^e < cve_{i+j+1}^e.$$

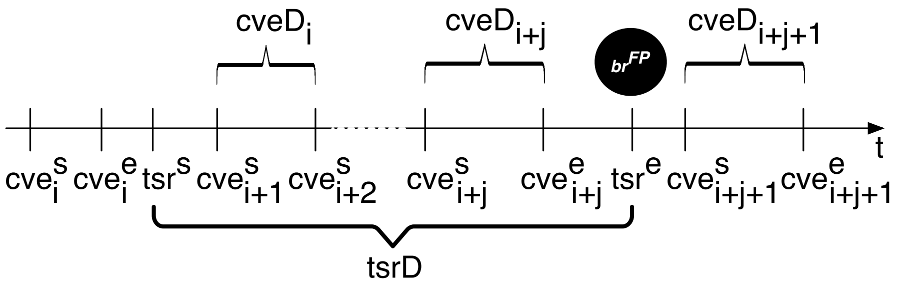


Figure 6-10 False positive basic test result (brC^{FP})

We count all false positive results using brC^{FP}.

- *Pseudo true negative basic test result counter* (brC^{PTN})*:* Similar to a true negative test result, a test produces a pseudo true negative result if it fails at a time when a control is violated. However, unlike a br^{TN}, a br^{PTN} is produced by a test only *partially* overlapping with the control violation event. There are two cases of partial overlapping to take into account:

1. *Failing test ends during control violation event:* A br^{PTN} is produced by a failing test which starts (tsr^s) prior to the start of the control violation event (cve^s). Furthermore, the test ends (tsr^e) after the violation events starts (cve^s) and before the control violation ends (cve^e):

$$br^{PTN} = tsr^s < cve_i^s \wedge cve_i^s \leq tsr^e \wedge tsr^e \leq cve_i^e.$$

   Consider, as an example, the following scenario: A test starts measuring available bandwidth of a virtual machine. Only after the test started, the limitation of bandwidth of the virtual machine is induced by a control violation event. Thus, while at the beginning of the test no control was violated, later during the test it was. If the measurement result in total determines that the available bandwidth was insufficient, then the test fails, producing a pseudo true negative result br^{PTN}.

2. *Failing test starts during control violation event:* A br^{PTN} is produced by a failing test which starts (tsr^s) after a control violation event starts (cve^s) and starts before the control violation event ends (cve^e). Further, the test only ends (tsr^e) after the violation events ends (cve^e):

$$br^{PTN} = cve_i^s \leq tsr\hat{}s \wedge tsr^s \leq cve_i^e < tsr^e.$$

Figure 6-11 shows a br^{PTN} where a correctly failing test ends during a control violation event and Figure 6-12 depicts the case where a correctly failing test starts during a control violation event. In Figure 6-11, note the dotted line between the start of the test (tsr^s) and the start of the violation event (cve^s). It indicates that a test can cover multiple control violation events. Similarly, in Figure 6-12, the dotted line between the end of the control violation event (cve^e) and the end of the test (tsr^e) indicates that the test may cover multiple control violation events.

If a test covers multiple cve, then this implies that a test takes longer to complete ($tsrD$) than the duration of the control violation event ($cveD_i$), that is, $tsrD > cveD_i$.

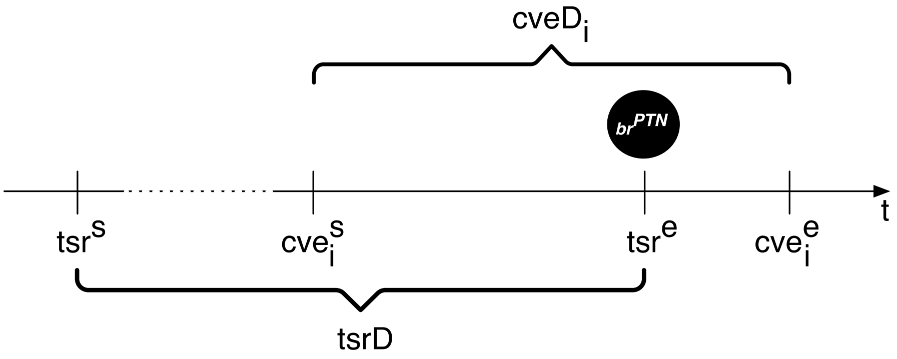


Figure 6-11 Pseudo true negative basic test result (brC^{PTN})

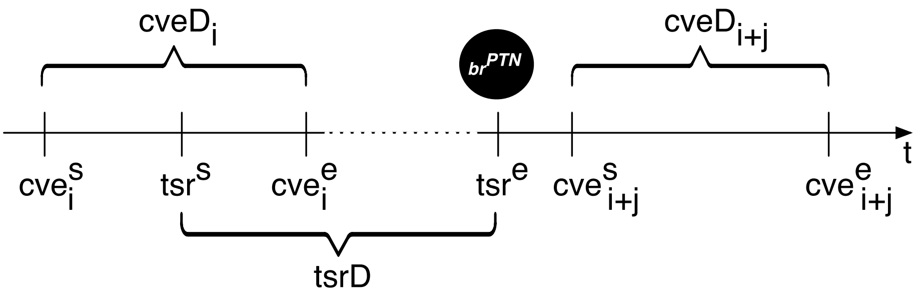


Figure 6-12 Pseudo true positive basic test result (br^{PTN})

Lastly, brC^{PTN} counts any occurrence of pseudo true negative test results.

- *Pseudo false positive basic test result counter* (brC^{PFP}): A test produces a pseudo false positive result if the test partially overlaps with a control violation event but incorrectly passes. This means that the definition of br^{PFP} is identical to br^{PTN}, the only difference being that the test result is positive. As in the case of a br^{PTN}, a

br^{PFP} can end during a control violation event or it can start during a control violation event. Also, a br^{PFP} may cover multiple control violation events. The number of occurrences of pseudo false positive results are counted using brC^{PFP}.

ACCURACY MEASURES BASED ON BRC

The previous paragraph introduced six evaluation measures based on the Basic-Result-Counter (brC) which serve to analyze the measurement results produced by a test-based measurement technique under evaluation during a control violation sequence. To summarize:

- True positive basic test result counter (brC^{TP}),
- true negative basic test result counter (brC^{TN}),
- false negative basic test result counter (brC^{FN}),
- false positive basic test result counter (brC^{FP}),
- pseudo true negative basic test result counter (brC^{PTN}), and
- pseudo false positive basic test result counter (brC^{PFP}).

These evaluation measures are used as input to compute accuracy measures. To that end, we draw on standard accuracy measures used in binary classification described by, e.g., (14), (15) and (16). Hereafter, it is described which specific measures are selected and how to interpret them to evaluate the accuracy of test-based measurement techniques.

- *Overall accuracy* (oac): The measure delineates the ratio between all correctly passed or failed tests ($brC^{TN} + brC^{PTN} + brC^{TP}$) and all observed test results ($brC^{TN} + brC^{PTN} + brC^{FN} + brC^{TP} + brC^{FP} + brC^{PFP}$). The overall accuracy permits to evaluate out of all observed measurement results of a test-based technique under evaluation, how many are correct results:

$$\mathrm{oac}^{\mathrm{brC}} = \frac{(\mathrm{brC}^{\mathrm{TN}} + \mathrm{brC}^{\mathrm{PTN}} + \mathrm{brC}^{\mathrm{TP}})}{(\mathrm{brC}^{\mathrm{TN}} + \mathrm{brC}^{\mathrm{PTN}} + \mathrm{brC}^{\mathrm{FN}} + \mathrm{brC}^{\mathrm{TP}} + \mathrm{brC}^{\mathrm{FP}} + \mathrm{brC}^{\mathrm{PFP}})}$$

- True negative rate (tnr): This measure delineates the proportion of correctly failed tests ($brC^{TN} + brC^{PTN}$) out of any test that should actually have failed ($brC^{TN} + brC^{PTN} + brC^{FP} + brC^{PFP}$). Using tnr, the ability of a test-based technique to correctly detect if a cloud services complies with a control or not can be analyzed:

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

- *True positive rate* (tpr): This measure describes the ratio between correctly passed tests (brC^{TP}) and all tests that were expected to pass ($brC^{TP} + brC^{FN}$). It permits to evaluate how well a test-based technique correctly indicates that a cloud service satisfies the control the test aims to check:

$$\mathrm{tpr}^{\mathrm{brC}} = \frac{\mathrm{brC}^{\mathrm{TP}}}{(\mathrm{brC}^{\mathrm{TP}} + \mathrm{brC}^{\mathrm{FN}})}$$

- *False negative rate* (fnr): This measure describes the ratio between incorrectly failed tests (brC^{FN}) and all tests that were expected to pass ($brC^{TP} + brC^{FN}$). Based on this measure, we can evaluate how often a test-based technique incorrectly suggests that a control is not fulfilled by a cloud service:

$$\mathrm{fnr}^{\mathrm{brC}} = \frac{\mathrm{brC}^{\mathrm{FN}}}{(\mathrm{brC}^{\mathrm{TP}} + \mathrm{brC}^{\mathrm{FN}})} = 1 - \mathrm{tpr}^{\mathrm{brC}}.$$

- *False positive rate* (fpr): This measure describes the ratio between incorrectly passed tests ($brC^{FP} + \mathrm{brC}^{\mathrm{PFP}}$) and all observed tests that actually should have failed ($brC^{TN} + brC^{PTN} + brC^{FP} + brC^{PFP}$). It permits to describe the proportion of a test-based technique's results which incorrectly suggest that a control of a cloud service is fulfilled:

$$\mathrm{fpr}^{\mathrm{brC}} = \frac{(\mathrm{brC}^{\mathrm{FP}} + \mathrm{brC}^{\mathrm{PFP}})}{(\mathrm{brC}^{\mathrm{TN}} + \mathrm{brC}^{\mathrm{PTN}} + \mathrm{brC}^{\mathrm{FP}} + \mathrm{brC}^{\mathrm{PFP}})} = = 1 - \mathrm{tnr}^{\mathrm{brC}}.$$

- *False discovery rate* (fdr): This measure captures the ratio between incorrectly passed tests ($brC^{FP} + brC^{PFP}$) and all test which passed ($brC^{FP} + brC^{TP} + brC^{PFP}$). This allows us to reason about how often (out of all observed positive test results) measurement results of a test-based technique should have indicated failure, that is, measurement results which incorrectly indicated that a cloud service satisfies a control:

$$\mathrm{fdr}^{\mathrm{brC}} = \frac{(\mathrm{brC}^{\mathrm{FP}} + \mathrm{brC}^{\mathrm{PFP}})}{(\mathrm{brC}^{\mathrm{FP}} + \mathrm{brC}^{\mathrm{TP}} + \mathrm{brC}^{\mathrm{PFP}})} = 1 - \mathrm{ppv}^{\mathrm{brC}}$$

- *Positive predictive value* (ppv): This measure delineates the ratio between correctly passed tests (brC^{TP}) and all test that passed ($brC^{TP} + brC^{FP} + brC^{PFP}$). Using this measure, it is possible to quantify the proportion of measurement results within all positive results which correctly suggest that a cloud service meets a control:

$$\mathrm{ppv}^{\mathrm{brC}} = \frac{brC^{TP}}{(\mathrm{brC}^{\mathrm{TP}} + \mathrm{brC}^{\mathrm{FP}} + \mathrm{brC}^{\mathrm{PFP}})} = 1 - \mathrm{fdr}^{\mathrm{brC}}.$$

- *False omission rate* (for): This measure describes the ratio between incorrectly failed tests (brC^{FN}) and all tests which failed ($brC^{TN} + brC^{PTN} + brC^{FN}$). This makes it is possible to describe the proportion of measurement results produced by a test-based technique that should have passed within all produced test result that failed:

$$\mathrm{for}^{\mathrm{brC}} = \frac{\mathrm{brC}^{\mathrm{FN}}}{(\mathrm{brC}^{\mathrm{TN}} + \mathrm{brC}^{\mathrm{PTN}} + \mathrm{brC}^{\mathrm{FN}})} = 1 - \mathrm{npv}^{\mathrm{brC}}.$$

- *Negative predictive value* (npv): This measure describes the ratio between correctly failed tests ($brC^{TN} + brC^{PTN}$) and all tests that failed ($brC^{TN} + brC^{PTN} + brC^{FN}$). This allows to capture the proportion of results produced by a test-based technique which correctly indicate that a cloud service does not meet a control:

$$\mathrm{npv}^{\mathrm{brC}} = \frac{(\mathrm{brC}^{\mathrm{TN}} + \mathrm{brC}^{\mathrm{PTN}})}{(\mathrm{brC}^{\mathrm{TN}} + \mathrm{brC}^{\mathrm{PTN}} + \mathrm{brC}^{\mathrm{FN}})} = 1 - \mathrm{for}^{\mathrm{brC}}.$$

PRECISION MEASURES BASED ON BRC

All accuracy measures based on evaluating basic test results (br), e.g., true negative rate (tnr), false positive rate (fpr), and negative predictive value (npv) have in common that they are *proportions*, that is, they provide the fraction of, e.g., correct test results of any observed test results. Thus we can construct confidence intervals for these proportions, that is, estimate the precision of these accuracy measures using interval estimates.

Consider, as an example, computing a confidence interval of 95% for npv^{brC}. This interval estimate allows statements such as *we are 95% confident that the* npv^{brC} *of a test-based technique under evaluation is contained in the interval*. This inference is valid with respect to the configuration of the test-based technique and the control violation sequence.

Continuing our example for npv^{brC}, we compute this interval estimate with

$$npv^{brC} \pm z_{95\%} \times se_{npv}.$$

$z_{95\%}$ is the value that separates the middle 95% of the area under the standard normal (or z) distribution, and se is the standard error which can be estimated with

$$\mathrm{se}_{\mathrm{npv}} = \sqrt{\widehat{\mathrm{npv}}^{\mathrm{brc}} \times \frac{(1 - \widehat{npv}^{brC})}{n}}.$$

$\widehat{npv}^{brC}$ makes an educated guess of npv proportion in the parent distribution. If no historical information on npv^{brC} of the parent distribution is available, then $\widehat{npv}^{brC} = 0.5$ can be chosen denoting the conservative option. Further, n is the sample size which in this example for npv^{brC} consists of any basic failed test result used to compute npv^{brC}, that is,

$$n = brC^{TN} + brC^{PTN} + brC^{FN}.$$

As stated above, the standard normal distribution is used to look up the value for $z_{95\%}$. This requires the sampling distribution of the proportion to be Gaussian. Determining the required sample size n, the margin of error $E_{npv}^{95\%} = z_{95\%} \times se$ is solved for the sample size $\tilde{n}$:

$$\tilde{n} = \frac{\mathrm{z}_{95\%} \times \widehat{\mathrm{npv}}^{\mathrm{brC}} \times \left(1 - \widehat{\mathrm{npv}}^{\mathrm{brC}}\right)}{\hat{E}^2}$$

where $\hat{E}$ delineates the desired margin of error.

Recall that in Section 6.3.2 and 6.3.3, the question was brought forward how many control violation events $|V|$ are needed to infer conclusions about the general accuracy of a test-based measurement technique under evaluation. Continuing the example for npv^{brC}, determining the required size of V can be formulated as an optimization problem:

$$minimize\ |V|$$

$$subject\ to\ \tilde{n}\ \le brC^{TN} + brC^{PTN} + brC^{FN}$$

Thus at least as many control violation events cve have to be induced as are required to observe $\tilde{n}$ test results. Following the above steps, interval estimates for the remaining accuracy measures, i.e., oac^{brC}, tnr^{brC}, tpr^{brC}, fnr^{brC}, fpr^{brC}, fdr^{brC}, ppv^{brC}, and for^{brC} introduced in Section 6.4.1 can be computed analogously.

6.4.2 FAILED-PASSED-SEQUENCE-COUNTER

This section describes how to estimate the accuracy and precision of a test-based measurement technique under evaluation using the Failed-Pass-Sequence-Counter metric ($fpsC$). To that end, the next section describes the evaluation of test results using three evaluation measures. Thereafter, it is described how we leverage these evaluation measures to compute accuracy and precision measures.

EVALUATION OF MEASUREMENT RESULTS

This section explains how to evaluate a test-based technique based on the Failed-Passed-Sequence-Counter metric ($fpsC$). Recall that $fpsC$ counts the occurrence of failed-passed-sequences (fps), it is a special sequence of basic test results which starts with a failed test and ends with the next passing test (see Section 6.1.1 for further detail). A fps aims at detecting temporal control violations, that is, control violations that persist for some time. In order to evaluate the measurement results of a test-based technique, we inspect if and how any fps overlaps with control violation events cve.

- *True negative fps* (fps^{TN}): A fps that consists of only correct basic test results, i.e., true negative test results (br^{TN}), pseudo true negative test results (br^{PTN}) and one final true positive test result (br^{TP}). A fps^{TN} starts (fps^{s}) after the last control violation event ends (cve^{e}_{i-1}) and starts before the next control violation event ends (cve^{e}_{i}). Furthermore, the fps^{TN} ends (fps^{e}) only after the next control violation ends (cve^{e}_{i}). Formally, we can define a true negative fps as follows:

$$fps^{TN} = cve^{e}_{i-1} \leq fps^{s} \wedge fps^{s} \leq cve^{e}_{i} \wedge cve^{e}_{i} < fps^{e}.$$

  Note that a fps^{TN} may cover multiple cve. Figure 6-13 shows an exemplary true negative fps whose first failed test produced a pseudo true negative result (br^{PTN}) which starts at tsr_j^{s}. This example fps^{TN} covers two control violation events, that is, cve_i and cve_{i+1}. $fpsC^{TN}$ counts the number of fps^{TN} observed during a control violation sequence.

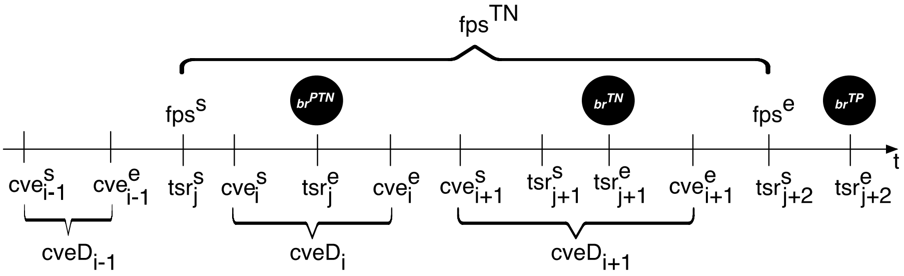


Figure 6-13 True negative failed-passed-sequence (fps^{TN})

  Note that a true negative fps which detects the first control violation event during experimental evaluation depicts a special case: If no previous cve exists, then the following, simplified definition of fps^{TN} applies:

$$fps^{TN} = fps^{s} \leq cve^{e}_{i} \wedge cve^{e}_{i} < fps^{e}.$$

- *False negative fps* (fps^{FN}): A fps that consists of at least one incorrect basic test result, i.e., false negative test results (br^{FN}) or false positive test result (br^{FP}) or both. A basic variant of an fps^{FN} is observed if any failed basic test results are false negatives and only the last test passes correctly. In this case, the fps starts after the last cve ends (cve^{e}_{i}) and ends (fps^{e}) before the next cve starts (cve^{s}_{i+1}):

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

Figure 6-14 shows this basic version of a fps^{FN}. We define $fpsC^{FN}$ which counts any occurrence of fps^{FN} observed during a control violation sequence.

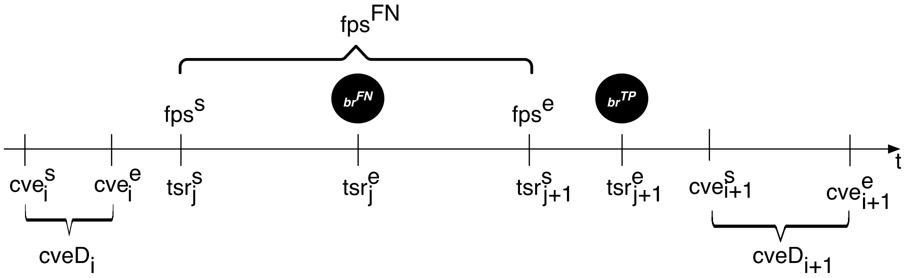


Figure 6-14 False negative fps

However, false negative fps may also contain true negative basic test results. This is the case if after a cve ended and before the next cve starts, that is, no control violation event is induced, basic results still incorrectly indicate a control violation. Figure 6-15 shows an example case of this error: After the control violation event cve_i ended at cve_i^e and before the next cve starts at cve_{i+1}^s, the test tsr_{j+1} produces a false negative test result at tsr_{j+1}^e.

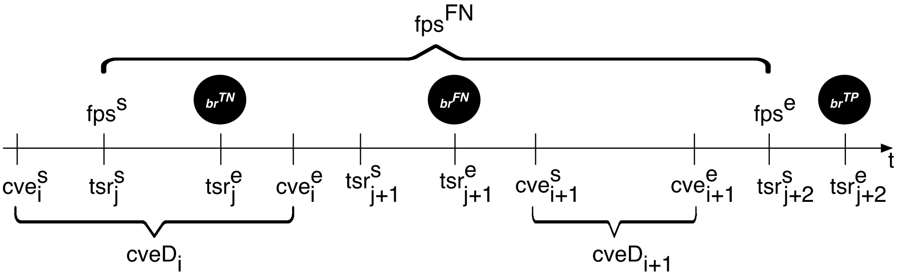


Figure 6-15 False negative failed-passed-sequence (fps^{FN}) with true negative and false negative basic test result (br^{TN} & br^{FN})

Complementary indicators for fps^{FN} are the false omission rate (for^{brC}) and negative predictive value (npv^{brC}). These accuracy measures are calculated using on basic test results (see Section 6.4.1). The more incorrect negative basic test results are observed during evaluation of a test-based technique, the higher for^{brC} and the lower npv^{brC}.

At last, the last test of an fps^{FN} can be a false positive, i.e., the last test result incorrectly indicates that the cloud services satisfies a control. Figure 6-16 shows one example of this error: After a test correctly failed at tsr_{j+1}^e, the next test incorrectly passes while the control is still violated, thereby producing a false positive test result (br^{FP}) at tsr_{j+2}^e.

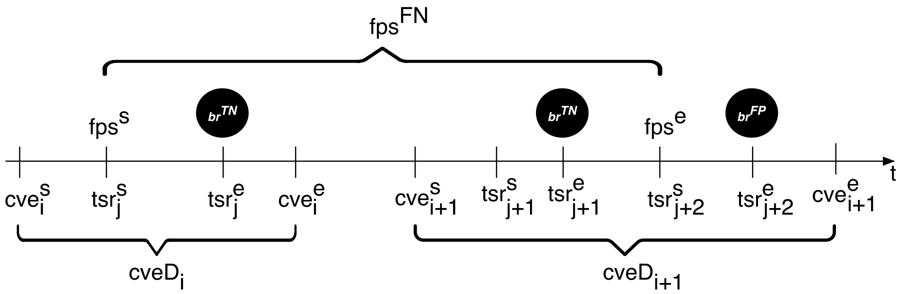


Figure 6-16 False negative failed-passed-sequence (fps^{FN}) *with false positive basic test result* (br^{FP})

As a complementary means to investigate this type of error, we can use of the positive predictive value (ppv^{brC}) and false discovery rate (fdr^{brC}) introduced in Section 6.4.1: The more incorrect positive basic test results are observed during evaluation, the higher fdr^{brC} and the lower ppv^{brC}.

- *False positive fps* (fps^{FP}): A fps indicates that a cloud service does not satisfy a control over time. Thus, a control violation event not detected by a test-based measurement technique is considered false positive fps. Figure 6-17 shows a cve that starts after the last fps ended (fps_j^e) and ends before the next fps starts (fps_{j+1}^s):

$$fps^{FP} = fps_j^e < cve^s \wedge cve^e < fps_{j+1}^s.$$

We use $fpsC^{FP}$ to count the occurrences of fps^{FP} during a control violation sequence.

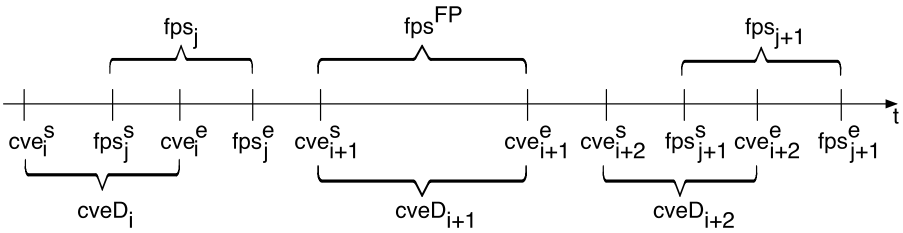


Figure 6-17 False positive fps

ACCURACY MEASURES BASED ON FPSC

The previous paragraphs introduced three evaluation measures derived from the Failed-Passed-Sequence-Counter ($fpsC$):

- True negative Failed-Passed-Sequence-Counter (fps^{TN}),
- false negative Failed-Passed-Sequence-Counter (fps^{FN}) and
- false positive Failed-Passed-Sequence-Counter (fps^{FP}).

These evaluation results are now used to calculate accuracy measures. To that end, analogous to the accuracy measures based on brC introduced in Section 6.4.1, standard measures used in binary classification are leveraged. The following paragraphs explain which measures are selected and how these measures can be used to interpret the accuracy of a test-based measurement technique under evaluation to identify temporal violations of controls.

- True negative rate (tnr): This measure describes the ratio between correctly detected control violation events ($fpsC^{TN}$) and all control violation events that were induced by the control violation sequence, that is, which could have been detected ($fpsC^{TN} + fpsC^{FP}$):

  $$tnr^{fpsC} = \frac{fpsC^{TN}}{(\mathrm{fpsC}^{\mathrm{TN}} + \mathrm{fpsC}^{\mathrm{FP}})} = 1 - fpr^{fpsC}.$$

  tnr^{fpsC} allows to evaluate how well a test-based measurement technique works in detecting intervals when a control is not satisfied by a cloud service.

- *False positive rate* (fpr): This measure describes how many control violation events were not detected ($fpsC^{FP}$) out of all events that could have potentially been detected ($fpsC^{TN} + fpsC^{FP}$):

  $$fpr^{fpsC} = \frac{fpsC^{FP}}{(fpsC^{TN} + fpsC^{FP})} = 1 - tnr^{fpsC}.$$

  Based on fpr^{fpsC}, the proportion can be described how many control violation events were missed by test-based technique under evaluation. It is the percentage of how many times the test-based technique failed to indicate that a control is not satisfied by a cloud service.

- *False omission rate* (for): This measure captures the ratio of incorrectly detected control violation events ($fpsC^{FN}$) and all control violation events that a test-based technique indicated ($fpsC^{TN} + fpsC^{FN}$):

  $$for^{fpsC} = \frac{fpsC^{FN}}{(fpsC^{TN} + fpsC^{FN})} = 1 - npv^{fpsC}.$$

  Using for^{fpsC}, it is possible to make statements about how often a test-based technique incorrectly suggested that a cloud service did not comply with a control for some time out of all detected control violation events.

- *Negative predictive value* (npv): This measure delineates the ratio between any correctly detected control violation event ($fpsC^{TN}$) and all detected control violation events ($fpsC^{TN} + fpsC^{FN}$):

  $$npv^{fpsC} = \frac{fpsC^{TN}}{(fpsC^{TN} + fpsC^{FN})} = 1 - for^{fpsC}.$$

  On the basis of npv^{fpsC}, it can be evaluated how many times a test-based measurement technique correctly indicated a control violation event out of all control violation events that the test-based technique suggested.

PRECISION MEASURES BASED ON (FPSC)

Analogous to the accuracy measures derived from basic test results, the accuracy measures tnr^{fpsC}, fpr^{fpsC}, for^{fpsC} and npv^{fpsC} can be treated as proportions. Therefore, we apply the same idea proposed in the previous section to calculate interval estimates for tnr^{fpsC}, fpr^{fpsC}, for^{fpsC} and npv^{fpsC} in order to infer general statements about the accuracy of a test-based measurement technique based on $fpsC$.

Note that there exists one important difference to the approach described in the previous section: At least as many control violation events $|V|$ have to be induced as are needed to observe $\tilde{n}$ fps during the control violation. Consider, as an example, that we want to construct a confidence interval for tnr^{fpsC}. The sample size n for tnr^{fpsC} consists of any control violation event which should have been detected by the test-based measurement technique, that is,

$$n = fpsC^{TN} + fpsC^{FP}.$$

The corresponding optimization problem to find the required sample size $\tilde{n}$ for tnr^{fpsC} thus can be formulated as follows:

$minimize\ |V|$

subject to $\tilde{n} \leq \mathrm{fpsC}^{\mathrm{TN}} + \mathrm{fpsC}^{\mathrm{FP}}$

Precision estimates for the remaining three accuracy measures, i.e., fpr^{fpsC}, for^{fpsC} and npv^{fpsC} can be computed analogously by following the above steps.

6.4.3 FAILED-PASSED-SEQUENCE-DURATION

This section describes how to estimate the accuracy and precision of a test-based technique based on measurement results produced by the universal test metric *Failed-Passed-Sequence-Duration* ($fpsD$). First, the different types of errors that an $fpsD$ may make when attempting to determine the duration, start and end of a control violation event are described. Thereafter, it is explained how to use these evaluation measures to estimate the accuracy and precision of a test-based measurement technique based on $fpsD$.

EVALUATION OF MEASUREMENT RESULTS

The following paragraphs describe how to evaluate a test-based measurement technique based on measurement results produced by the *Failed-Passed-Sequence-Duration* test metric ($fpsD$). Recall that $fpsD$ captures the time (e.g., in milliseconds) 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 (see Section 6.1.1).

- *Duration error of true negative fpsD* ($efpsD^{TN}$): Having observed a true negative fps, the difference between the duration of the fps, i.e., $fpsD = fps^e - fps^s$ and the duration $cveD$ of any control violation events which is covered by the fps is calculated. Figure 6-18 shows that a fps^{TN} may cover multiple cve, however, it can, at most, cover all cve contained in the sequence V of the control violation sequence:

  $$\mathrm{fpsD^{TN}} = \mathrm{fpsD^{TN}} - \sum_{i=1}^{|V|} \mathrm{cveD_i} \ .$$

  Note that we do *not* calculate the absolute difference between $fpsD$ and covered $cveD$. This permits us to determine whether a $fpsD$ overestimates or underestimates the duration of a control violation event: In case of $efpsD^{TN} > 0$, then the $fpsD$ overestimates the duration of covered control violation events (Figure 6-18). Otherwise, if $efpsD^{TN} < 0$, then $fpsD$ underestimates the duration of the control violation event (Figure 6-19). Lastly, if $efpsD^{TN} = 0$, the $fpsD$ and the duration of the covered control violation events are equal.

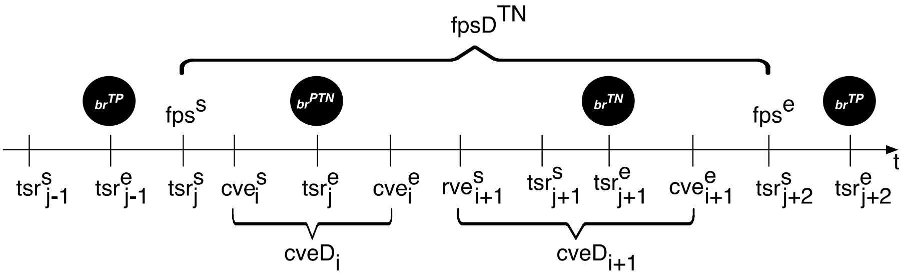


Figure 6-18 True negative Failed-Passed-Sequence-Duration (fps^{TN}) *which overestimates total duration of* cve_i *and* cve_{i+1}

Furthermore, the relative error that a fps makes when estimating the duration of covered control violation events is calculate as follows:

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

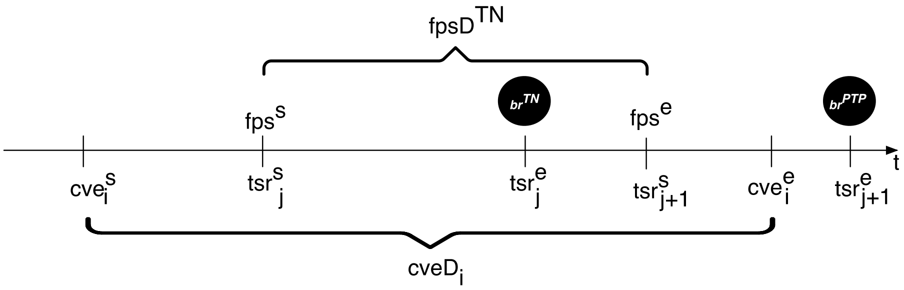


Figure 6-19 True negative Failed-Passed-Sequence-Duration (fps^{TN}) *which underestimates duration of* cve_i

- *Pre-duration error of true negative fpsD* $(efpsD_{pre}^{TN})$: Until now, we focused our error definition on the estimated duration of control violation events provided by a true negative $fpsD$. However, as Figure 6-20 illustrates, the start of a $fpsD^{TN}$ which estimates the start of the control violation event can be inaccurate, i.e., $cve^s < fps^s$. In order to capture this error, we compute the difference between the start of a fps, i.e., the start of the first failed test which detected a control violation event (fps^s), and the start of the control violation event (cve^s):

$$efpsD_{pre}^{TN} = fps^s - cve^s.$$

In case of $efpsD_{pre}^{TN} > 0$, then the fps starts only after the cve starts. Note that this case implies that the first failed test of the $fpsD^{TN}$ produced a true negative test result (br^{TN}). Further, if $efpsD_{pre}^{TN} < 0$, then the fps starts before the cve starts. This case, in turn, implies that the first test produced a pseudo true negative test result (br^{PTN}).

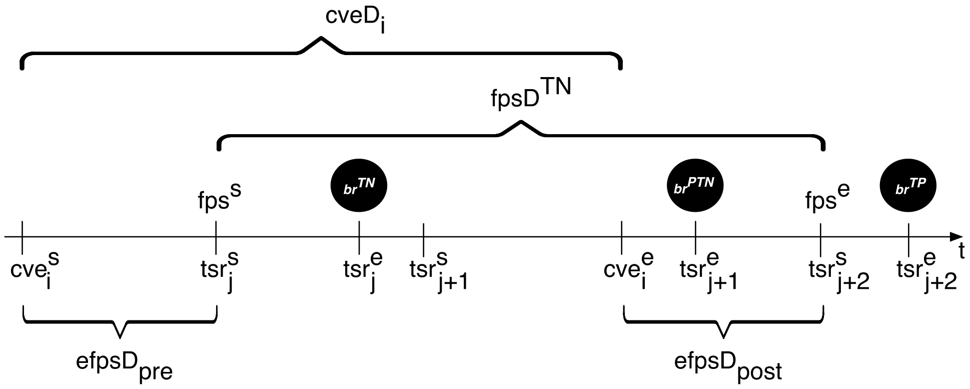


Figure 6-20 True negative Failed-Passed-Sequence-Duration (fps^{TN}) *with* $efpsD_{pre}^{TN} > 0$ *and* $efpsD_{post}^{TN} > 0$.

- *Post-duration error on true negative* $fpsD$ ($efpsD_{post}^{TN}$): Recall that the last basic test result of a true negative fps is always a true positive basic test result. This means that a fps^{TN} by definition only ends after the control violation event ends. Figure 6-20 shows $efpsD_{post}^{TN}$ which is the resulting error the last test result of a $fpsD^{TN}$ makes when determining the end of a control violation event. Describing this error, the difference between the end of a control violation event (cve^e) and the end of the fps, i.e., the start of the last test which passed is computed:

$$efpsD_{post}^{TN} = fps^e - cve^e.$$

- *Duration error of false negative* psD ($efpsD^{FN}$): If a false negative fps is observed, then the entire duration of that fps is considered to be erroneous because it incorrectly indicates a duration of a control violation event. Figure 6-21 shows a $fpsD^{FN}$ which is defined as follows:

$$efpsD^{FN} = fps^e - fps^s.$$

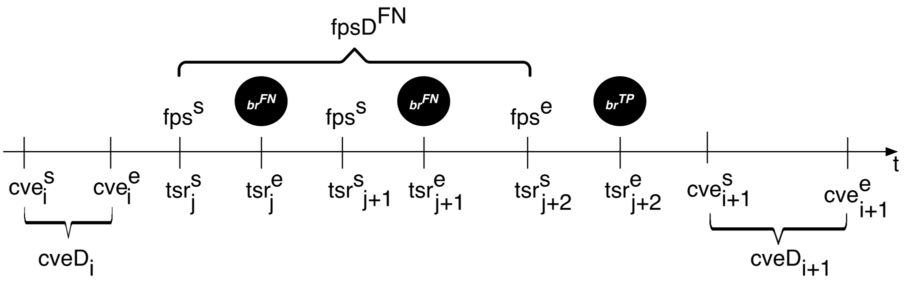


Figure 6-21 False negative Failed-Passed-Sequence-Duration ($fpsD^{FN}$)

- *Duration error of false positive* $fpsD$ ($efpsD^{FP}$): If a control violation event is not detected by a fps at all, then this missed cve is considered a false positive fps.

Consequently, the duration of a false positive fps equals the duration of the missed control violation event (Figure 6-22):

$$efpsD^{FP} = cve^{e} - cve^{s}.$$

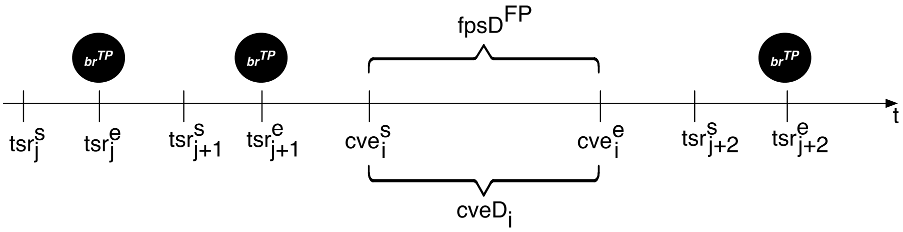


Figure 6-22 caption False positive Failed-Passed-Sequence-Duration ($fpsD^{FP}$)

ACCURACY MEASURES BASED ON EFPSD

In the previous section, we introduced five error types derived from the Failed-Passed-Sequence-Duration ($fpsD$) test metric:

- Duration error of true negative Failed-Passed-Sequence-Duration ($efpsD^{TN}$),
- pre-duration error of true negative Failed-Passed-Sequence-Duration ($efpsD^{TN}_{pre}$),
- post-duration error of true negative Failed-Passed-Sequence-Duration ($efpsD^{TN}_{post}$),
- duration error of false negative Failed-Passed-Sequence-Duration ($efpsD^{FN}$), and
- duration error of false positive Failed-Passed-Sequence-Duration ($efpsD^{FP}$).

When evaluating a test-based measurement technique, then instances of any of the above errors may be observed. These observations for each type of error on $fpsD$ can be treated as separate distributions: After having executed a control violation sequence and the test-based measurement technique under evaluation, it can be expected to obtain at most five distributions. However, in practice, a test-based technique may not produce any incorrect test results, i.e., neither br^{FN} nor br^{FP}. This means that neither instances of $efpsD^{FN}$ not instances of $efpsD^{FP}$ are observed. However, a test-based technique which does not make any error on estimating the total duration, the start and the end of any control violation event is rather unlikely. The reason for this is that not observing any instance of $efpsD^{TN}$, $efpsD^{TN}_{pre}$, or $efpsD^{TN}_{post}$ requires the test-based technique to always perfectly estimate duration, start and end of any control violation event. Thus, it is reasonable to expect to observe at least three distributions after having evaluated a test-based measurement technique, i.e., $efpsD^{TN}$, $efpsD^{TN}_{pre}$ and $efpsD^{TN}_{post}$.

In order to estimate the accuracy of a test-based technique when measuring temporal control violations (e.g., in milliseconds), the arithmetic mean ($\bar{x}$) for each of the observed distributions is computed. For example, to compute the arithmetic mean for $efpsD^{TN}$, we add any instances i of $efpsD^{TN}$ contained in the sequence $EFPSD^{TN}$ and divide by the number of elements in $EFPSD^{TN}$:

$$\bar{x}^{TN} = \frac{(efpsD_1^{TN} + efpsD_2^{TN} + \cdots + efpsD_i^{TN})}{|EFPSD^{TN}|}.$$

Using $\bar{x}^{TN}$, we can describe the average error a $fpsD^{TN}$ makes when estimating the duration of a control violation event. Calculation and interpretation of the remaining four error types is analogous.

As a complementary measure, also the median ($\hat{x}$) is computed which is the middle value of an ordered list. The median is helpful when values of, e.g., $EFPSD_{pre}^{TN}$ do not increase arithmetically, i.e., if the difference between consecutive values of an ordered list is not constant. Consider, as an example, having observed $EFPSD_{pre}^{TN} = \langle -8, -5\,, 10 \rangle$. The mean is $\bar{x}_{pre}^{TN} = -1$ while median tells us $\hat{x}_{pre}^{TN} = -5$.

PRECISION MEASURES BASED ON EFPSD

Describing the precision of a test-based measurement technique under evaluation, the following statistics are computed:

- *Standard deviation* (sd): This statistic measures the dispersion of values within a distribution. Drawing on the example from the previous paragraph, the standard deviation of the values in $EFPSD^{TN}$ describe how far values spread around its mean:

  $$sd_{TN} = \sqrt{\frac{1}{|EFPSD^{TN}|} \times ((efpsD_1^{TN} - \bar{x}_{TN})^2 + \cdots + \left(efpsD_i^{TN} - \bar{x}_{TN}\right)^2)}$$

  Using sd, it is possible to describe the variation of the different types of error which a test-based technique makes when measuring the duration of control violation events. Furthermore, the sd can also be used to calculate the standard error of the mean which is needed to calculate confidence intervals which is explained in the following paragraph.

- *Confidence Interval for the sample mean*: In total, five types of errors were presented which a $fpsD$ may make when measuring the duration of a control violation event, e.g., $efpsD^{TN}$ and $efpsD^{FP}$. For each of these error types, the mean $\bar{x}$ of the observed distribution is computed serving as an accuracy measure. In order to make a general statement about the precision of a test-based measurement technique, we can construct a confidence interval for each mean.

As an example, consider $efpsD^{TN}$, which captures the mean error that a test-based technique makes when determining the duration of a control violation event: A confidence interval on this mean permits us statements such as *we are 99% confident that the average error of a test-based* measurement *technique* – with respect to the technique's and control violation configuration – *makes when estimating the duration of a control violation event is contained in the interval*. This estimate can be obtained as follows:

$$\bar{x}_{TN} \pm t_{99\%} \times se_{\bar{x}}.$$

$t_{99\%}$ is the value that separates the middle 99% of the area under the t-Distribution and se is the standard error. se can be estimated as follows:

$$se_{\bar{x}} = \frac{sd_{TN}}{\sqrt{n}}.$$

In context of the above example, the sample size n is the number of observed true negative fps and sd is the standard deviation. In order to determine the required sample size $\tilde{n}$, the desired margin of error $\hat{E}$ is solved for the sample size $\tilde{n}$, that is,

$$\tilde{n} = \frac{\sigma^2 \times t_{99\%}^2}{\hat{E}^2}.$$

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 accuracy of a test-based measurement techniques based on the mean of, e.g., $\bar{x}_{TN}$ requires inducing a minimum number of control violation events. In our example for $efpsD^{TN}$, the minimum size of V can be obtained by solving the following optimization problem:

$minimize\ |V|$

$subject\ to\ \tilde{n} \leq \mathrm{fpsD}^{\mathrm{TN}}$

This means that at least as many control violation events need to be induced which are needed to observe $\tilde{n}\ fpsD^{TN}$. Using these above steps, interval estimates for the means of $efpsD_{pre}^{TN}$, $efpsD_{post}^{TN}$, $efpsD^{FN}$, and $efpsD^{FP}$ can be calculated analogously.

Also, the minimum and maximum (min & max) are computed, that is, the smallest and largest value for any type of error that was observed during evaluation of a test-based measurement technique. Using these statistics, most extreme errors that a test-based technique makes when measuring duration of control violation events can be described. Furthermore, comparing min and max to the standard deviation can help identifying if the measurement results produced by the test-based technique during evaluation contain outliers.

6.4.4 CUMULATIVE-FAILED-PASSED-SEQUENCE-DURATION

In this section, we describe how to determine the accuracy of a test-based measurement technique based on the universal test metric *Failed-Passed-Sequence-Cumulative-Duration* ($cfpsD$). Hereafter, first the three evaluation measures $cfpsD^{TN}$, $cfpsD^{FN}$, and $cfpsD^{FP}$ are introduced which are derived from $fpsD^{TN}$, $fpsD^{FN}$, and $fpsD^{FP}$ observed during evaluation, respectively. Thereafter, it is explained how these evaluation measures can be leveraged to estimate the accuracy of a test-based technique under evaluation.

EVALUATION OF MEASUREMENT RESULTS

This section describes how to evaluate a test-based measurement technique based on the *Failed-Passed-Sequence-Cumulative-Duration test metric* ($cfpsD$). Recall that this metric accumulates the value of any $fpsD$ (e.g., in milliseconds) observed within a specified period of time. This allows to determine if a cloud service satisfies a control with temporal constraints within that period (see Section 6.1.1).

- *True negative cfpsD ($cfpsD^{TN}$):* Each value of a true negative $fpsD$ observed during evaluation of the test-based measurement technique is added, i.e.,

  $$cfpsD^{TN} = fpsD_1^{TN} + fpsD_2^{TN} + \cdots + fpsD_i^{TN}.$$

  This measure returns the total measured duration of correctly detected control violation events.

- *False negative cfpsD* ($cfpsD^{FN}$): This evaluation measure holds the sum of any false negative *fpsD* which was produced by the test-based measurement technique under evaluation:

  $$cfpsD^{FN} = fpsD_1^{FN} + fpsD_2^{FN} + \cdots + fpsD_i^{FN}.$$

  $cfpsD^{FN}$ captures the total measured duration of control violation events which the test-based technique incorrectly indicated.

- *False positive cfpsD* ($cfpsD^{FP}$): The sum of any false positive $fpsD$ which was produced by the test-based technique under evaluation is computed by this metric:

$$cfpsD^{FP} = fpsD_1^{FP} + fpsD_2^{FP} + \cdots + fpsD_i^{FP}.$$

Using $cfpsD^{FP}$, the total duration of control violation events that were *not* detected by the test-based measurement technique under evaluation can be described.

ACCURACY MEASURES BASED ON (CFPSD)

The previous three paragraphs introduced the following three evaluation measures:

- True negative Cumulative-Failed-Passed-Sequence-Duration ($cfpsD^{TN}$)
- false negative Cumulative-Failed-Passed-Sequence-Duration ($cfpsD^{FN}$), and
- false positive Cumulative-Failed-Passed-Sequence-Duration ($cfpsD^{FP}$).

In order to determine the overall accuracy of a test-based technique within a predefined period of time, that is, within the control violation sequence, the following three accuracy measures can be used:

- *Duration error of true negative cfpsD* ($ecfpsD^{TN}$): This measure computes the difference between the cumulative duration of true negative fpsD and the total duration of any control violation event $cve \in V$:

$$ecfpsD^{TN} = cfpsD^{TN} - \sum_{i=0}^{|V|} cveD_i.$$

The accuracy measure $ecfpsD^{TN}$ permits to describe if a test-based technique overestimates or underestimate the accumulated duration of control violations within a specified period of time. If the test-based technique overestimates the total duration of violated controls, then $ecfpsD^{TN} > 0$. Otherwise, if the test-based technique underestimates the total duration of violated controls, then $ecfpsD^{TN} < 0$. Lastly, if $ecfpsD^{TN} = 0$, then the duration measured by the test-based technique perfectly matches the total duration of control violation events.

Furthermore, the ratio between $ecfpsD^{TN}$ and the total duration of control violation events is computed:

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

Using $ecfpsD_{rel}^{TN}$, it is possible to describe the relative measurement error that a test-based technique makes when determining the total time during which a cloud service does not comply with a control.

- *Duration error of false negative cfpsD* ($ecfpsD^{FN}$): The total duration of false negative $fpsD$ that a test-based technique suggested is identical to the duration error of false negative $cfpsD$, that is, $cfpsD^{FN} = ecfpsD^{FN}$. However, the absolute total duration of a test-based technique's measurement results incorrectly indicating temporary control violation provides only limited information because it lacks context. Therefore, we also compute the ratio between $ecfpsD^{FN}$ and the total amount of time during which the test-based technique indicated that the cloud service does not satisfy a control $(ecfpsD^{FN} + ecfpsD^{TN})$:

  $$ecfpsD_{rel}^{FN} = \frac{cfpsD^{FN}}{(cfpsD^{FN} + cfpsD^{TN})}.$$

  Based on $ecfpsD_{rel}^{FN}$, we can make statements about the proportion of detected temporary control violation which – out of the total duration of control violation events – was incorrect.

- *Duration error of false positive $cfpsD$ ($ecfpsD^{FP}$):* The total duration of false positive $fpsD$ is identical to the duration error of false positive $cfpsD$, i.e., $cfpsD^{FP} = ecfpsD^{FP}$. Yet $ecfpsD^{FP}$ as an absolute value only provides the total amount of time where the test-based technique was we expected to detect temporary control violation events but, in fact, it did not. In order to be able to assess the meaning of $ecfpsD^{FP}$, we relate it to total duration of control violation events as follows:

  $$ecfpsD_{rel}^{FP} = \frac{cfpsD^{FP}}{\sum_{i=1}^{|V|} cveD_i}$$

  where $ecfpsD_{rel}^{FP}$ describes the proportion of control violation events' duration which remained undetected in total.

PRECISION MEASURES BASED ON CFPSD

Recall the definition of *precision* presented in Section 6.1.2: Precision refers to closeness of agreement between successively measured values which implies that precision measures need at least two measured values as input. Since $ecfpsD^{TN}$, $ecfpsD^{FN}$, and $ecfpsD^{FP}$ are exactly calculated once after experimental evaluation of a test-based measurement technique, the concept of precision is not applicable drawing on $cfpsD$.

6.5 IMPLEMENTATION AND EXAMPLE EVALUATION

This section presents an example scenario in which we apply our method to evaluate and compare measurement results produced by a test-based measurement technique. The next section describes the components of our experimental setup. Thereafter, we present a scenario in which cloud service providers seek to evaluate tests to support continuous certification of cloud services according to controls related to the properties availability and security.

6.5.1 SETUP AND ENVIRONMENT

This section outlines the experimental setup used to evaluate measurement results produced by the test-based technique. We begin with the cloud service which is subject to testing. Then we briefly describe the control violation framework which is used to manipulate properties of the cloud services under test so that it does not comply with one or more controls as well as the test-based measurement technique. Finally, we present the evaluation engine which is used to computes the accuracy and precision measures presented in Section 6.4.

CLOUD SERVICES UNDER TEST

The cloud service under test consists of an instance of IaaS provided by OpenStack Mitaka[18] on top of which an Apache[19] web server is running. The virtual machine is equipped with 2 VCPUs and 4 GB RAM and running Ubuntu 16.04 server.

CONTROL VIOLATION FRAMEWORK

In order to trigger control violation events, a lightweight framework has been developed in Java which permits to repeatedly manipulate properties of a cloud service under test over time so that the service does not satisfy one or more controls for some time (for further detail see Section 6.3). The framework is extensible allowing to add novel control violation and multiple control violation sequences can be executed concurrently.

Each control violation event is persisted, including start and end time of each event, event duration as well as current iteration. This data serves as the reference which is later used by the evaluation engine (see paragraph below) to evaluate the accuracy and precision of measurement results produced by a test-based measurement technique. The control violation

---

[18] https://www.openstack.org/software/mitaka/

[19] https://httpd.apache.org/

framework is deployed on a designated virtual machine, attached to the identical tenant network as the cloud service under test.

CONTINUOUS TEST-BASED MEASUREMENT TECHNIQUE

The test is implemented following the framework introduced in Section 4 of Deliverable 3.2. The test is deployed on an external host, attached to a different network than the cloud services under test.

EVALUATION ENGINE

This component calculates accuracy and precision measures described in Section 6.4 as well as test and control violation statistics. For that purpose, the *Apache Commons Math* library is used. The evaluation engine is implemented in Java and runs locally on a personal computer and uses the control violation sequence's data and produced test results as input.

6.5.2 CONTINUOUSLY TESTING SECURE COMMUNICATION CONFIGURATION

In this scenario, we consider a cloud service provider who, at the same time, acts a cloud service customer. This means that the provider offers a SaaS application to customers for whose delivery he leverages another cloud provider offering platform services (PaaS). Thus, components such as web server, data bases, and load balancer are supplied and maintained by the PaaS provider. Therefore, the SaaS provider cannot directly access the underlying applications and components but only has access to the necessary APIs. As a result, the PaaS provider is responsible to provide secure communication configurations which includes secure configuration of Transport Layer Security (TLS) used by the web server component of the SaaS application to deliver websites via HTTPS.

We assume that the SaaS provider seeks certification of his application according to controls which relate to property *secure communication configuration*. Examples for such cloud-specific controls are *KRY-02 Encryption of data for transmission (transport encryption)* of the Cloud Computing Compliance Controls Catalogue (BSI C5) (7), *EKM-03: Encryption & Key Management Sensitive Data Protection* of CSA's Cloud Control Matrix (CCM) (1), and *A.14.1.2 Securing application services on public networks* of ISO/IEC 27001:2013 (17).

In order to support certification of his SaaS application, the provider want to utilize a continuous test-based measurement technique and configure it in such a way that it indicates as accurately as possible if the secure communication configuration property of his SaaS application does not hold. This implies that the test-based technique should ideally detect any violation of the secure communication configuration property and the number of false positive measurement results produced by the technique should be as low as possible. Furthermore, if an insecure communication configuration is detected, then the SaaS provider seeks a test configuration which as accurately as possible detects how long the PaaS provider needs to remedy vulnerable communication configurations.

ALTERNATIVE TEST CONFIGURATIONS

In order to analyze TLS configurations of our cloud service under test, we leverage the tool *sslyze*[20]. Inspecting the output of sslyze permits to, e.g., find out whether the web server offers to communicate via known vulnerable cipher suites. If the web server does offer support for vulnerable cipher suites, then the secure communications configuration property of the cloud service under test does not hold which, in turn, leads to a violation of certificates' controls relating to this property.

The SaaS provider within our scenario can select one of the following three different candidate configurations for the test *TLSTest*:

- $TLSTest^{[0,10]}$: Each execution of *TLSTest* is triggered randomly in the interval $[0,10]$ after the last test completed.
- $TLSTest^{[0,30]}$: Each execution of *TLSTest* is triggered randomly in the interval $[0,30]$ after the last test completed.
- $TLSTest^{[0,60]}$:Each execution of *TLSTest* is triggered randomly in the interval $[0,60]$ after the last test completed.

No additional offset between test executions is configured while the number of successive iterations for all three *TLSTest* variants is set to infinity. Further, only measurement results produced during the control violation sequence are considered for evaluation.

CONTROL VIOLATION CONFIGURATION

For each TLSTest variant, we triggered 1000 vulnerable TLS configurations of the cloud service under test to evaluate the three candidate configurations of TLSTest. These vulnerable TLS configurations consist of altering the web server configuration such that it supports TLS communication using the weak cipher suite TLS_RSA_WITH_DES_CBC_SHA. Each event of an

---

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

insecure TLS configuration lasted at least 60 seconds plus selecting $[0,30]$ seconds at random. The interval between consecutive vulnerable configuration events lasted at least 120 seconds plus selecting $[0,60]$ seconds at random. Table 6-1 summarizes the control violation sequence statistics observed during experimental evaluation of *TLSTest*$^{[0,10]}$ *TLSTest*$^{[0,30]}$ and *TLSTest*$^{[0,60]}$.

Table 6-1 Summary of control violation sequence statistics for TLSTest

Sequence statistic (sec)	$V^{\textbf{TLSTest}}$		
	[0,10]	**[0,30]**	**[0,60]**
ccveD	75050.77	74817.15	75477.49
$mean_{cveD}$	75.10	74.82	75.48
sd_{cveD}	8.90	8.97	9.06
min_{cveD}	60.01	60.02	60.02
max_{cveD}	90.04	90.10	90.03

TEST STATISTICS

The measurement results produced by TLSTest are shown in Table 6-2: They consist of any results observed for each of the universal test metrics presented in Section 6.1.1. Moreover, the total number of executed tests ($tsrC$) as well as the mean ($mean_{tsr}$), standard deviation (sd_{tsr}), min (min_{tsr}) and (max_{tsr}) duration of tests are included. Note that for each TLSTest variant, we only observed a single value for false positive $fpsD$ (i.e., $fpsC^{FP} = 1$) and thus we cannot compute average (x_{FP}), median ($median_{fpsD^{FP}}$), and standard deviation (sd_{FP}) for *TLSTest*$^{[0,10]}$ *TLSTest*$^{[0,30]}$ and *TLSTest*$^{[0,60]}$. The corresponding fields of Table 6-2 are marked as *not applicable (na)*.

ACCURACY AND PRECISION OF TLSTEST

This section presents the results of evaluating the accuracy and precision of *TLSTest*$^{[0,10]}$ *TLSTest*$^{[0,30]}$ and *TLSTest*$^{[0,60]}$.

- *Accuracy and precision based on Basic-Result-Counter (brC):* Table 6-3 shows the results of evaluating *TLSTest*$^{[0,10]}$, *TLSTest*$^{[0,30]}$ and *TLSTest*$^{[0,60]}$ on evaluation measures which are derived from the Basic-Result-Counter (brC) test metric. According to our scenario, the SaaS provider wants to select a configuration of TLSTest which produces the least number of false positive basic test results (brC^{FP}): *TLSTest*$^{[0,10]}$ produced the highest number of brC^{FP}, followed by *TLSTest*$^{[0,30]}$ and *TLSTest*$^{[0,60]}$ (Table 6-2). However, solely relying on the absolute counts of brC^{FP} is misleading because *TLSTest*$^{[0,10]}$ executed more than twice as many tests *TLSTest*$^{[0,30]}$. Thus, we have to make use of the accuracy and precision measures introduced in Section 6.4.1 which relate brC^{FP} to the remaining measurement results produced by the test-based measurement technique. These inlcude: Overall accuracy (oac^{brC}), true negative rate (tnr^{brC}), false positive rate (fpr^{brC}), false discovery rate (fdr^{brC}) and positive predictive value (ppv^{brC}).

TLSTest$^{[0,10]}$ has the lowest overall accuracy (98.24%) and the lowest true negative rate (97.06%). Further, *TLSTest*$^{[0,10]}$ has the highest false discovery rate (1.55%), followed by *TLSTest*$^{[0,60]}$ (1.46%) and *TLSTest*$^{[0,30]}$ (1.34%). However, the most suitable accuracy measure in context of our scenario is the false positive rate since it captures the ratio between incorrectly passed tests and all test that were expected to fail: *TLSTest*$^{[0,10]}$ has the highest fpr (2.94%), followed by *TLSTest*$^{[0,60]}$ (2.84%) and *TLSTest*$^{[0,30]}$ (2.64%). As a consequence, the SaaS provider selects *TLSTest*$^{[0,30]}$ if he only relies on the accuracy derived from the brC test metric.

- *Accuracy and precision based on Failed-Passed-Sequence-Counter ($fpsC$):* Table 6-4 presents the results of evaluating *TLSTest*$^{[0,10]}$, *TLSTest*$^{[0,30]}$ and *TLSTest*$^{[0,60]}$ based on the universal test metrics Failed-Passed-Sequence-Counter ($fpsC$). Recall that the SaaS provider within our scenario seeks to configure TLSTest such that it produces the lowest number of false positive results possible. In context of accuracy and precision measures based on the $fpsC$ test metric, we therefore select the false positive rate (fpr^{fpsC}) and the true negative rate (tnr^{fpsC}) – as defined in Section 6.4.2 – to evaluate the variants of TLSTest since they tell us – out of all events that should have been detected – how many control violation events were correctly detected (tnr^{fpsC}) and how many control violation events were not detected (fpr^{fpsC}).

  Despite each of the TLSTest variants only producing a single false positive fps (see Table 6-2), *TLSTest*$^{[0,60]}$ has the lowest fdr (0.1%) and the highest tnr (99.9%) because *TLSTest*$^{[0,60]}$ produced a higher number of true negative fps (969) than *TLSTest*$^{[0,10]}$ (871) and *TLSTest*$^{[0,30]}$ (893). Hence, if the SaaS provider only draws on the accuracy based on the $fpsC$ test metric, then he chooses *TLSTest*$^{[0,60]}$.

- *Accuracy and precision based on Failed-Passed-Sequence-Duration ($fpsD$):* Evaluating *TLSTest*$^{[0,10]}$, *TLSTest*$^{[0,30]}$, and *TLSTest*$^{[0,60]}$ based on the Failed-Passed-Sequence-Duration ($fpsD$) produces the results shown in Table 6-5. Since only a single value for false positive $fpsD$ (i.e., $fpsC^{FP} = 1$, see Table 6-2) for each TLSTest variant has been observed, we cannot calculate mean, median, standard deviation (sd) and margin of error ($E^{95\%}$) of $efpsD^{FP}$ for *TLSTest*$^{[0,10]}$, *TLSTest*$^{[0,30]}$, and *TLSTest*$^{[0,60]}$. This is indicated by marking the corresponding fields of Table 6-5 as *not applicable (na)*.

Besides choosing a configuration for TLSTest which produces the lowest false positive results, our example SaaS provider prefers the TLSTest variant which as accurately as possible estimates how long it takes the PaaS provider to remedy a detected, vulnerable communication configuration. In other word: The SaaS provider prefers a configuration of TLSTest which most accurately estimates the duration of a correctly detected control violation event.

Figure 6-23 shows three box plots which capture the variation of relative duration error of true negative fps ($efpsD_{rel}^{TN}$) for the three TLSTest variants: It is obvious that the relative error each test of *TLSTest*$^{[0,60]}$ makes when estimating the duration has the highest mean (dashed green line inside the box, 22.96%), median (solid red line inside the box, 20.56%) as well as the highest variability. Further, on average, *TLSTest*$^{[0,10]}$ produces true negative fps having the lowest relative error $efpsD_{rel}^{TN}$ when estimating the duration of a vulnerable communication configuration event (4.56%), followed by *TLSTest*$^{[0,30]}$ (11.33%). Hence, in context of our scenario, the SaaS provider prefers *TLSTest*$^{[0,10]}$ since this configuration of TLSTest provides the most accurate estimate of how long it takes the PaaS provider to fix a vulnerable TLS configuration.

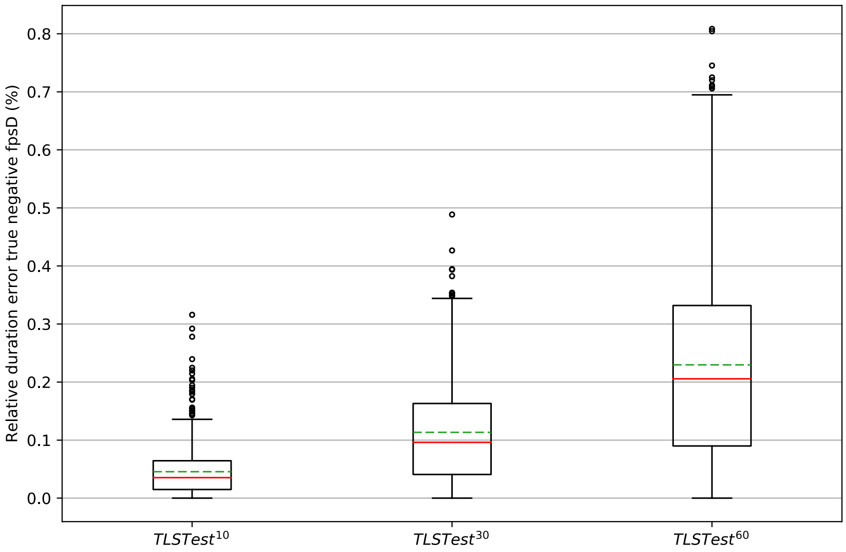


Figure 6-23 Relative duration error of fpsD ($efpsD_{rel}^{TN}$) of TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$, and TLSTest$^{[0,60]}$

- *Accuracy and precision based on cumulative-Failed-Passed-Sequence-Duration ($cfpsD$):* Table 6-6 shows the results of evaluating *TLSTest*$^{[0,10]}$, *TLSTest*$^{[0,30]}$, and *TLSTest*$^{[0,60]}$ using the universal test metric cumulative-Failed-Passed-Sequence-Duration ($cfpsD$). The results of the total duration error of true negative fps ($ecfpsD^{TN}$) show that all evaluated variants of TLSTest underestimate the accumulated duration of vulnerable TLS configuration events. Drawing on $ecfpsD^{TN}$, the most accurate result is produced by *TLSTest*$^{[0,60]}$ (-2232.09 seconds), followed by *TLSTest*$^{[0,30]}$ (-7476.24 seconds) and *TLSTest*$^{[0,10]}$ (-9974.22 seconds). However, the accumulated duration of true negative fps is outside the scope of our example scenario since the SaaS provider's focus lies on correctly detecting temporary vulnerable TLS configurations and estimating their duration. Therefore, the accumulated duration of fps and thus the accumulated error of fps does *not* affect the decision of the SaaS provider which variant of TLSTest to select.

CONCLUSION

The SaaS provider in our example scenario favors *TLSTest*$^{[0,60]}$ because the accuracy and precision measures $efpsC$ indicate that it has the highest number of correctly detected control violations, that is, true negative fps. One may argue that this conclusion is flawed because *TLSTest*$^{[0,10]}$ is more accurate in estimating the duration of a vulnerable TLS configuration event (see accuracy and precision measures $efpsD$). However, we presume that it is more important to the SaaS provider in our scenario that the continuous test-based measurement technique detects the number of occurrences of control violations most accurately than it is to most accurately estimate the duration of those violations correctly detected. Also, although outside the scope of our example evaluation scenario, comparing the accuracy of the TLSTest variants based on the cumulative error of true negative fps ($ecfpsD^{TN}$) would further foster our conclusion because $TLSTest^{[0,60]}$ produces the lowest value for $ecfpsD^{TN}$.

Table 6-2 Summary of test statistics of TLSTest

Test statistic		**TLSTest**		
		[0,10]	**[0,30]**	**[0,60]**
	tsrC	34801	13771	7332
tsr (sec)	$mean_{tsr}$	1.50	1.40	1.38
	sd_{tsr}	0.59	0.62	0.46
	min_{tsr}	0.10	0.10	0.10
	max_{tsr}	19.73	19.39	19.18
brC	brC^{TP}	22484	9024	4793
	brC^{FP}	8	5	3
	brC^{TN}	11585	4504	2410
	brC^{FN}	260	83	39
	brC^{PTN}	106	33	18
	brC^{PFP}	346	118	68
fpsC	$fpsC^{TN}$	871	893	969
	$fpsC^{FN}$	184	110	30

	$fpsC^{FP}$	1	1	1
fpsD (sec)	$mean_{TN}$	74.78	75.41	75.59
	sd_{fpsD}^{TN}	9.94	13.93	22.96
	$median_{fpsD}^{TN}$	74.55	75.43	75.30
	min_{fpsD}^{TN}	41.75	39.64	18.20
	max_{fpsD}^{TN}	97.99	114.76	138.10
	$mean_{FN}$	52.32	73.46	69.66
	sd_{FN}	32.52	19.29	29.55
	$median_{fpsD}^{FN}$	66.17	73.54	65.87
	min_{fpsD}^{FN}	0.40	13.09	24.27
	max_{fpsD}^{FN}	96.23	114.18	143.17
	$mean_{FP}$	*na*	*na*	*na*
	sd_{FP}	*na*	*na*	*na*
	$median_{fpsD}^{FP}$	*na*	*na*	*na*
	min_{fpsD}^{FP}	87.02	73.02	84.02
	max_{fpsD}^{FP}	87.02	73.02	84.02
cfpsD (sec)	*TN*	65136.55	67340.92	73245.40
	FN	9627.13	8080.22	2089.73
	FP	87.02	73.02	84.02

Table 6-3: Evaluation of TLSTest to test secure communication configuration of $SaaS^{OS}$ *based on the basic result counter (brC) test metric*

Test		**TLSTest**		
statistic		**[0,10]**	**[0,30]**	**[0,60]**
ebrC (%)	*oac*	98.24	98.50	98.50
	$E_{95\%,oac}$	0.14	0.20	0.28
	tnr	97.06	97.36	97.16
	$E_{95\%,tnr}$	0.30	0.46	0.65
	tpr	98.86	99.09	99.19
	$E_{95\%,tpr}$	0.14	0.20	0.25
	fnr	2.22	1.83	1.61
	$E_{95\%,fnr}$	0.26	0.38	0.49
	fpr	2.94	2.64	2.84
	$E_{95\%,fpr}$	0.30	0.46	0.65
	fdr	1.55	1.34	1.46
	$E_{95\%,fdr}$	0.16	0.24	0.34
	ppv	98.45	98.66	98.54
	$E_{95\%,ppv}$	0.16	0.24	0.34
	for	2.18	1.80	1.58
	$E_{95\%,for}$	0.26	0.38	0.49
	npv	97.82	98.20	98.42
	$E_{95\%,npv}$	0.26	0.38	0.49

Table 6-4: Evaluation of TLSTest to test secure communication configuration of SaaS^{OS} based on the failed-passed-sequence Counter (fpsC) test metric

Test statistic		**TLSTest**		
		[0,10]	**[0,30]**	**[0,60]**
efpsC (%)	*tnr*	99.89	99.89	99.9
	$E_{95\%,tnr}$	0.22	0.22	0.20
	fpr	0.11	0.11	0.10
	$E_{95\%,fpr}$	0.22	0.22	0.20
	for	17.44	10.97	3.0
	$E_{95\%,for}$	2.29	1.93	1.06
	npv	82.56	89.03	97.0
	$E_{95\%,npv}$	2.29	1.93	1.06

Table 6-5: Evaluation of TLSTest to test secure communication configuration of SaaS^{OS} based on the failed-passed-sequence Duration (fpsD) test metric

Test statistic		**TLSTest**		
		[0,10]	**[0,30]**	**[0,60]**
efpsD^{TN}**(ms)**	*mean*	-52	644	151
	median	254	603	-442
	sd	4508	10465	20991
	min	-22201	-40073	-51054
	max	11552	25906	51510

	$E^{95\%}$	300	687	1323
efpsD^{TN,rel}(%)	*mean*	4.56	11.33	22.96
	median	3.54	9.62	20.56
	sd	4.15	8.68	16.45
	min	0.01	0.01	0.01
	max	31.58	48.86	80.83
	$E^{95\%}$	0.28	0.57	1.04
efpsD^{TN,pre}(ms)	*mean*	4677	10587	21490
	median	4030	9308	19383
	sd	3759	7682	14688
	min	-1582	-774	44
	max	25739	45251	61204
	$E^{95\%}$	250	505	925
efpsD^{TN,post}(ms)	*mean*	4624	11230	21641
	median	4217	10154	19407
	sd	2502	6999	14098
	min	18	31	18
	max	15350	29288	58807
	$E^{95\%}$	166	460	889
efpsD^{FN}(ms)	*mean*	52321	73457	69658
	median	66173	73536.5	65874.5

	sd	32517	19295	29548
	min	396	13087	24270
	max	96231	114183	143168
	$E^{95\%}$	4730	3646	11033
$efpsD^{FP}$(ms)	*mean*	*na*	*na*	*na*
	median	*na*	*na*	*na*
	sd	*na*	*na*	*na*
	min	87024	73022	84022
	max	87024	73022	84022
	$E^{95\%}$	*na*	*na*	*na*

Table 6-6: Evaluation of TLSTest to test secure communication configuration of $SaaS^{OS}$ based on the cumulative failed-passed-sequence Duration (cfpsD) test metric

Test statistic		**TLSTest**		
		[0,10]	**[0,30]**	**[0,60]**
$ecfpsD^{TN}$	*TN*(ms)	-9914223	-7476238	-2232089
	TN(%)	13.21	9.99	2.96
$ecfpsD^{FN}$	*FN*(ms)	9627129	8080222	2089733
	FN(%)	12.88	10.71	2.77
$ecfpsD^{FP}$	*FP*(ms)	87024	73022	84022
	FP(%)	0.12	0.10	0.11

7 CONCLUSION

In this deliverable, first a tool chain was presented which implements continuous cloud security audits to support cloud certification. This tool chain draws on existing tools available as background in the EU-SEC project, including:

- *Clouditor*, an example of a continuous test-based measurement technique,
- *CTP API and CTP Server*, specification and example of an objective evaluation application,
- *STARWatch*, an application to help organizations manage compliance with CSA STAR through self-assessment, as well as
- *Slipstream*, a brokerage service that facilitates deployment of evidence as well as claim storage.

Thereafter, a process was described how to integrate the tool chain with existing cloud services. The steps of this process include:

- Selection of global integration strategy for measurement techniques,
- deployment of tool chain
- discovery of cloud service,
- derivation of feasible measurement techniques,
- selection of suitable metrics,
- starting execution of measurement techniques, and
- adaption of measurement techniques at operation time.

Finally, an approach was presented which allows to evaluate accuracy and precision of measurement results produced by continuous test-based measurement techniques. To that end, first the universal test metrics brC, $fpsC$, $fpsD$, and $cfpsD$ were introduced and it was defined what accuracy and precision mean with regard to cloud service certification. Then, the evaluation process was presented and here the notion of control violation sequences was introduced. Events of these sequences manipulate a cloud service property such that the service does not adhere to one or more controls anymore. These control violation sequences establish the reference values which we treat as the ground truth and which we use to evaluate the accuracy and precision of a technique's measurement results. Further, at the heart of the evaluation process are so-called evaluation measures which are derived on the basis of the universal the metrics. These measures are derived through comparing events of a control violation sequence with the measurement results computed according to these metrics. These measures permit statements about, e.g., the average error a test-based technique makes when measuring the duration of control violation events. Finally, an example evaluation was presented where it is shown how - according to some scenario-specific assumptions - a cloud provider can select the most suited configuration for a particular test-based technique.

8 BIBLIOGRAPHY

1. **Cloud Security Alliance (CSA).** Cloud Control Matrix: Security Controls Framework for Cloud Providers & Consumers. 2015.

2. **International Organization for Standardization (ISO).** Information technology -- Cloud computing -- Service level agreement (SLA) framework -- Part 1: Overview and concepts. 2016.

3. **Cloud Security Alliance (CSA).** Custom Applications and IaaS Trends 2017. [Online] 2017. https://downloads.cloudsecurityalliance.org/ assets/survey/custom-applications-and-iaas-trends-2017.pdf.

4. **Stephanow, Philipp and Banse, Christian.** Evaluating the performance of continuous test-based cloud service certification. *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).* 2017.

5. **Stephanow, Philipp and Khajehmoogahi, Koosha.** Towards continuous security certification of Software-as-a-Service applications using web application testing techniques. *31th IEEE International Conference on Advanced Information Networking and Applications (AINA).* 2017, pp. 931-938.

6. **Stephanow-Gierach, Philipp.** Continuous test-based certification of cloud services. Phd Thesis, 2018.

7. **Bundesamt für Informationssicherheit (BSI).** Cloud Computing Compliance Controls Catalogue (C5). Available: https://www.bsi.bund.de/ SharedDocs/Downloads/EN/BSI/Publications/CloudComputing/ ComplianceControlsCatalogue-Cloud_Computing-C5.pdf?__blob= publicationFile&v=3, 2016.

8. **Hughes, Ifan and Hase, Thomas.** Measurements and their uncertainties: A practical guide to modern error analysis. s.l. : Oxford University Press, 2010.

9. **Taylor, Barry N and Kuyatt, Chris E.** Guidelines for evaluating and expressing the uncertainty of NIST measurement results. s.l. : US Department of Commerce, Technology Administration, and National Institute of Standards and Technology (NIST), 1994.

10. **Owen, Art B.** Monte Carlo theory, methods and examples. Available: http://statweb.stanford.edu/~owen/mc/, 2013.

11. **Box, George EP and Hunter, William Gordon and Hunter, J Stuart.** Statistics for experimenters: An introduction to design, data analysis, and model building. s.l. : JSTOR, 1978. Vol. 1.

12. **Freedman, David and Pisani, Robert and Purves, Roger.** Statistics – 4th Edition. s.l. : W.W. Norton & Company, 2007.

13. **Freedman, David A.** Statistical models: Theory and practice. s.l. : Cambridge University Press, 2009.

14. **Fawcett, Tom.** An introduction to ROC analysis. *Pattern recognition letters.* 2006.

15. **Powers, David Martin.** Evaluation: From Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation. *Bioinfo Publications.* 2011.

16. **Stehman, Stephen V.** Selecting and interpreting measures of thematic classification accuracy. *Remote Sensing of Environment.* 1997, Vol. 62, 1, pp. 77--89.

17. **International Organization for Standardization (ISO).** ISO/IEC 27001:2013 Information technology -- Security techniques -- Information security management systems -- Requirements. 2013.

APPENDIX A

EU-SEC CA API

This is audit-api serves evidences for a continuous audit. In the EU-Sec project such evidences are collected and evaluated to determine the compliance status based on controls from the ccm.

More information: http://www.sec-cert.eu/

Contact Info: contact@sec-cert.eu

Version: 1.0.4

BasePath:/euseccaapi

All rights reserved

Access

APIKey KeyParamName:api_key KeyInQuery:false KeyInHeader:true

HTTP Basic Authentication

Methods

[Jump to Models]

Table of Contents

get /{scope}/identityfederation/admins/

post /{scope}/identityfederation/data/access

get /{scope}/identityfederation/{userId}/logins

get /{scope}/identityfederation/{userId}/auth

get /{scope}/identityfederation/{userId}/groups

CaApiScope

get /scope/

CaApiDatalocation

Up

get /{scope}/datalocation/{objectId}/geolocation/

Returns location the ISO 3166-1 alpha-2 country code of the location of the data of the object (getDataLocationGeolocation)

Retrieves the data location of an object. Returns location the ISO 3166-1 alpha-2 country code of the location of the data of the object. Based on CCM-STA-05.

Path parameters

objectId (required)

Path Parameter — ID of either objectId on SaaS level or storeageId on lower level

scope (required)

Path Parameter — Scope of the service

Return type

GeoLocation

Example data

Content-Type: application/json

{

"countryCode" : "at"

}

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

application/json

Responses

200

successful operation GeoLocation

405

Invalid input

Up

get /{scope}/datalocation/{objectId}/storage/

Returns persistence information for a particular data object by its Id (getDataLocationStorage)

Depending on the kind of storage this call returns an identifier of the particular storage entity. E.g database name, RDS id, Harddrive, SMB location etc. If stored on multiple services all are returned. Based on CCM-GRM-02. This requires each logical object to be traceable to its physical persistence capabilities.

Path parameters

objectId (required)

Path Parameter — ID of data object to return

scope (required)

Path Parameter — Scope of service

Return type

LocationStorageResponse

Example data

Content-Type: application/json

```
{
"storages" : [ {
"uri" : "i-0434c5582f2853d0c",
"type" : "service",
"description" : "AWS EC2 insctance"
}, {
"uri" : "vol-04b6088c76eb68a73",
"type" : "service",
"description" : "AWS EBS instance"
}, {
"uri" : "jdbc:mysql://192.168.0.10/SuperDB",
"type" : "database"
} ]
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

application/json

Responses

200

successful operation LocationStorageResponse

405

Invalid input

CaApiEncryption

Up

get /{scope}/encryption/{objectId}/

Retrieves the encryption info of an object. (getEncryptionInfo)

Based on CCM-EKM-04. Retrieves the encryption info of an object. Propper interpretation has to be performed by the audit tool.

Path parameters

objectId (required)

Path Parameter — ID of either objectId on SaaS level or storeageId on lower level

scope (required)

Path Parameter — Scope of the service

Return type

EncryptionStorageResponse

Example data

Content-Type: application/json

```
{
"keyOrigin" : [ {
"keyOriginUri" : "hsm://secret.datacenterX",
"type" : "hsm",
"description" : "Supersecure HSM"
}, {
```

```
"keyOriginUri" : "smb://key.pem",
"type" : "localKeyFile",
"description" : "Used for AES-256 enc."
} ]
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

application/json

Responses

200

successful operation EncryptionStorageResponse

405

Invalid input

CaApiIam

Up

get /{scope}/identityfederation/admins/

Returns a list of administrators (getAdmins)

Based on CCM-IAM-12. Reads out all administrators of the application and returns them.

Path parameters

scope (required)

Path Parameter — Scope of the service

Return type

AdminResponse

Example data

Content-Type: application/json

```
{

"admins" : [ "adminXYZ", "root", "caixaAuth" ]

}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

application/json

Responses

200

successful operation AdminResponse

405

Invalid input

Up

post /{scope}/identityfederation/data/access

Checks whether a user has a certain access to an object. (getObjectAccess)

Checks whether a user has a certain access to an object.

Path parameters

scope (required)

Path Parameter — Scope of the service

Request body

request AccessRequest (required)

Body Parameter — request object

Return type

AccessResponse

Example data

Content-Type: application/json

```
{

"allowed" : true

}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

application/json

Responses

200

successful operation AccessResponse

405

Invalid input

Up

get /{scope}/identityfederation/{userId}/logins

Returns a list of the last logins of a user (getUserAccesses)

Based on CCM-IAM-12. Whenever a user logs in into the application this kind access gets logged. This call returns the last accesses of a particular user based on existing logs.

Path parameters

userId (required)

Path Parameter — ID of user

scope (required)

Path Parameter — Scope of the service

Query parameters

from (optional)

Query Parameter — from date

limit (optional)

Query Parameter — Limits the number of retuned values

Return type

LoginResponse

Example data

Content-Type: application/json

```
{
"loginTimes" : [ "2005-08-15T15:52:01+0000" ]
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

application/json

Responses

200

successful operation LoginResponse

405

Invalid input

Up

get /{scope}/identityfederation/{userId}/auth

Returns the authentication type of a user. E.g password, two-factor (getUserAuthType)

Based on CCM-IAM-12. Reads out a particular users authentication settings and returns them. Propper interpretation has to be performed by the audit tool.

Path parameters

userId (required)

Path Parameter — ID of user

scope (required)

Path Parameter — Scope of the service

Return type

AdminAuth

Example data

Content-Type: application/json

```
{
"description" : "description",
"type" : { }
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

application/json

Responses

200

successful operation AdminAuth

405

Invalid input

Up

get /{scope}/identityfederation/{userId}/groups

Returns the groups of a user (getUserOrganisation)

Based on CCM-IAM-12. Depending on the implementation a group can be e.g a unix group, organisation, role etc.

Path parameters

userId (required)

Path Parameter — ID of user

scope (required)

Path Parameter — Scope of the service

Return type

GroupsResponse

Example data

Content-Type: application/json

{

"groups" : ["root", "awsEc2Full", "users"]

}

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

application/json

Responses

200

successful operation GroupsResponse

405

Invalid input

CaApiScope

Up

get /scope/

Returns all socpes of the cloud service (getScope)

Returns the available scopes for the cloud service. The scope corresponds often with the layers of the cloud service architecture like IaaS, PaaS, SaaS.

Return type

ScopeResponse

Example data

Content-Type: application/json

```
{

"scopes" : [ "SaaS", "PasS", "IaaS" ]

}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

application/json

Responses

200

successful operation ScopeResponse

405

Invalid input

Models

[Jump to Methods]

Table of Contents

LocationStorageResponse storages -

LoginResponse -

ScopeResponse -

StorageType -

AccessRequest - Up

userId

String

objectId

String

access

AccessType

AccessResponse - Up

allowed

Boolean true if access is allowd and false if its not.

AccessType - Up

AdminAuth - Up

type

AuthType

description (optional)

String

AdminResponse - Up

admins

array[String]

AuthType - Up

EncryptionStorageResponse - Up

keyOrigin

array[EncryptionStorageResponse_keyOrigin]

EncryptionStorageResponse_keyOrigin - Up

Object contains information about the key origin depending on type

keyOriginUri

String

type

KeyOriginType

description (optional)

String Comments on technical details of the key origin.

GeoLocation - Up

countryCode

String ISO 3166-1 alpha-2 country code

GroupsResponse - Up

groups

array[String]

KeyOriginType - Up

LocationStorageResponse - Up

storages

array[LocationStorageResponse_storages]

LocationStorageResponse_storages - Up

storageUri

String

type

StorageType

description (optional)

String Description should have comments that specify information, that will be stored in the description field. E.g., based on 5.6. call, description should contain technical details like database type with its version.

LoginResponse - Up

loginTimes

array[String] List of iso date time. Y-m-d\TH:i:sO ISO-8601

ScopeResponse - Up

scopes

array[String]

StorageType - Up