EUSEC
EU SECURITY CERTIFICATION

EUROPEAN SECURITY CERTIFICATION FRAMEWORK

D3.2 ARCHITECTURE AND TOOLS FOR AUDITING

V 1.0

PROJECT NUMBER: 731845

PROJECT TITLE: EU-SEC

DUE DATE: 31/12/2017	DELIVERY DATE: 27/12/2017
AUTHOR:	PARTNERS CONTRIBUTED:
Philipp Stephanow-Gierach	NIXU, CSA
DISSEMINATION LEVEL:*	NATURE OF THE DELIVERABLE:**
PU	R
INTERNAL REVIEWERS: SixSq, SI-MPA	

*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 develops and implements a unified way to configure test-based measurement techniques, the crucial part necessary to implement continuous security audits. To that end, a domain specific language (DSL) called *ConTest* is developed which allows to rigorously define continuous test-based measurements, that is, what is measured and how. While ConTest is agnostic to specific implementations of test-based measurement techniques, it also serves as a starting point from which specific configurations of a measurement technique can be automatically generated. That way, ConTest ensures that the configuration of a test-based measurement technique producing some measurement results adheres to the domain concepts defined for continuous test-based measurements.

In order to develop ConTest, the building blocks of continuous test-based measurements are described. These building blocks serve as the basis to identify and scope the required domain constructs to design ConTest. It is outlined how *Clouditor*, one exemplary tool to implement test-based measurement techniques, applies these building blocks to implement exemplary continuous test scenarios. After having analysed the building blocks, ConTest is formally defined using the Extended-Backus-Naur Form (EBNF) which is a notation used to define context-free grammars.

ConTest is implemented using the language development tool *XText*. It is shown how ConTest can be used as a starting point to generate exemplary configurations for specific implementation of test-based measurement techniques provided by Clouditor.

DISCLAIMER

The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Communities. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
AWS RDS	Amazon Web Service Rational Database Service
CCM	Cloud Control Matrix
CSA	Cloud Security Alliance
CSS	Cascading Style Sheets (CSS)
CT	continuous test
DSL	Domain-specific language
EBNF	Extented Backus-Naur Form
EMF	Eclipse Modeling Framework
EU-SEC	European Security Certification Framework
GPL	General Purpose Language
HTML	Hypertext Markup Language
HTTPS	HTTP Secure
IaaS	Infrastructure-as-a-Service
ICMP	Internet Control Message Protocol
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
MPS	Meta Programming System
OS	Operating System
OWASP	Open Web Application Security Project,
PaaS	Platform-as-a-Service

SaaS	Software-as-a-Service
SLO	Service level objective
SQL	Structured Query Language
SQLI	SQL injection
SQO	Service qualitative objective
SSH	Secure Shell
TC	test case
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TP	testing precondition
TS	test suite
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language

TABLE OF CONTENTS

List of Figures

1 INTRODUCTION

While development of standards for cloud services and certification schemes is well under way (e.g. BSI C5 (1), CSA STAR (2)), suitable techniques supporting continuous, i.e. repeated and automated certification of cloud services is subject to ongoing research and development. Such techniques are required because when applying traditional certification to cloud services, the following discrepancy surfaces (e.g. Khan and Malluhi (3), Ko et al. (4), Sunyaev and Schneider (5) and Cimato et al. (6)): Conducting a certification process is a discrete task, that is, the process produces a certificate at some point in time and this certificate is then considered valid for some time, usually in the range from one to three years (7). Put differently: Traditional certification assumes that during the period where a certificate is valid, any other security audit of the cloud service will produce identical results (8). However, a cloud service may change over time where the changes are hard to predict or detect by a cloud service customer (9). These changes may lead to the cloud service *not* fulfilling one or more certificate's control objectives, thus rendering the certificate invalid. Therefore, the assumption of stability underlying traditional certification does not hold in context of cloud services. Cloud service certification thus requires a different approach which uses *continuous security audits* to detect ongoing changes of a cloud service during operation and assesses their impact on satisfaction of certificates' control objectives (10) (11).

Continuous security audits are based on a chain of techniques which allow to automatically and repeatedly produce and reason about *evidence*. Evidence delineates observable information which serves as basic elements to check whether a cloud service possesses some set of properties and thus complies with one or more control objectives defined by a certification schema.

Naturally, *continuous security audits* are not to be understood in a strict mathematical sense: No matter how sophisticated the techniques to produce and reason about evidence, producing evidence will always be – in a strict mathematical sense – a discrete task that occurs at some point time. The term *continuous security audit* is used to describe *automated* and *repeated* production of and reasoning about evidence instances which is conducted by a third party and which occurs multiple orders of magnitude more frequently when compared to traditional certification (e.g. checking security attribute satisfaction every minutes instead of every year).

1.1 SCOPE AND OBJECTIVE

Certification aims at increasing a customer's trust towards a cloud service and enabling comparability between different cloud services. Thus when leveraging the concept of continuous security audits to automatically and repeatedly produce and reason about evidence, it is vital to *unambiguously describe how evidence is produced and processed* because those results are used to decide if one or more control objectives are satisfied at a certain point in time.

1.1.1 MOTIVATION AND PROBLEM STATEMENT

Before describing the problem of unambiguous evidence production in detail, it is necessary to recall and concretize the underlying concepts how evidence is produced and processed which have been introduced in Section 3 of Deliverable 2.2. Figure 1 provides an overview of the different concepts when implemented by a concrete chain of techniques support continuous security audits: *Evidence production technique* provide, e.g. by using tests, some form of *evidence*, e.g. supported TLS cipher suites of a cloud service's public endpoint (Step 1). Instance of evidence are then processed by a *metric*, i.e. a function which takes as input evidence and outputs measurement results (Step 2). A *measurement technique* therefore consists of at least one evidence production technique and one metric. In context of the TLS cipher suite example, a concrete metric may inspect the list of supported cipher suites and check whether it only contains those of a predefined whitelist which are considered secure. A result produced by that metric either indicates that all supported cipher suites are secure (*isSecure*) or are not secure (*isNotSecure*). *Measurement results* of this exemplary metric thus follow the nominal scale. After having been produced, measurement results are forwarded to *control objective evaluation*. Satisfaction of a control objective, again, can be understood as a function which takes a measurement result as input and outputs a *claim*, that is, a result indicating whether a control objective holds. Recall that in the TLS cipher suite example, measurement results follow the nominal scale. Thus, by definition, the control objective which is evaluated using these measurement results has to be derived from a Service Qualitative Objective (i.e., a SQO).

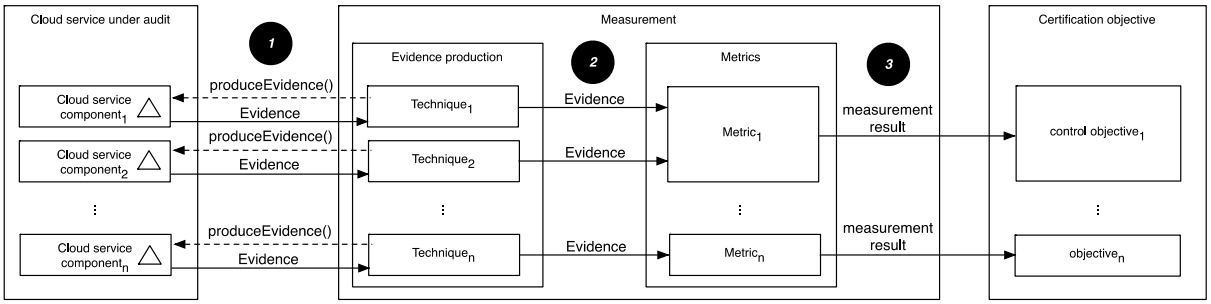


Figure 1: Continuous security audits

Given the terminology of the previous paragraph, the goal of this deliverable can be summarized as follows: An approach is needed to represent the semantics of measurement techniques' configuration in an unambiguous and comparable manner.

Consider, for example, the control RB-02 Capacity management – monitoring of the Cloud Computing Compliance Controls Catalogue (1) issued by The German Federal Office for Information Security (BSI). This control states that

> "*Technical and organisational safeguards for the monitoring and provisioning and de-provisioning of cloud services are defined. Thus, the cloud provider ensures that resources are provided and/or services are rendered according to the contractual agreements and that compliance with the service level agreements is ensured.*".

Lets further assume that the availability of two SaaS applications is compared where both SaaS providers have defined the identical SLA regarding availability (e.g. 99.9999% per year) and both providers claim to fulfil control RB-02 of BSI C5. In order to check whether their claims are true, one continuous audit strategy may consist of continuously testing both applications to detect potential outages, i.e. periods where the service is unavailable. Yet the measurement technique used for each application differ: One service's availability is continuously tested by simply pinging its endpoint every minute, i.e. measures the time delta between sending ICMP Echo Request to a publicly reachable host and receiving ICMP Time Reply packets. These measured round trip times serve as instances of evidence which are – in order to compute a metric – then compared with expected ones to determine the outcome of the test, i.e. whether the SaaS application is available or not. The other application is tested by issuing specially crafted calls to its RESTful API every 30 seconds and comparing the returned JSON object (i.e., the evidence) with the predefined, expected object to compute a measurement results indicating if the returned JSON is correct or not (i.e., the metric). It is obvious that two different measurement techniques produce measurement results that differ in semantics. These results are used to support evaluation whether the respective SaaS application complies with the above control objective. As a consequence, the conclusions drawn based on the differing

measurement results produced for the two exemplary SaaS applications cannot be directly compared.

The above example illustrates that *unambiguous configuration of measurement techniques* is needed to provide for comparability of measurement results generated as part of the continuous security audit. Furthermore, unambiguous definitions of how to collect and reason about evidence can also be leveraged to guide the design of future measurement techniques aiming to continuously produce evidence and compute measurement results. This can be understood as one central step of standardizing how to define the meaning of measurement results produced as part of continuous security audits. Thus it becomes possible to ensure that measurement results produced by some measurement techniques developed at some point in the future also follow the same rigorous definition of measurement techniques semantics.

Note that the above paragraph implies that measurement techniques' configurations have to be complete, that is, contain all information required to configure specific measurement technique. At the same time, these configurations have to be general representations of measurements which are agnostic to specific implementations of measurement technique.

1.1.2 APPROACH AND SCOPE

One approach to unified, rigorous representation of configurations of measurement techniques consists of the development of a domain-specific language (DSL) which is based on the notion of *formal languages* drawing on precise mathematical definitions (12). While this DSL has to be agnostic to implementation of measurement techniques and thus provides a way to define measurement result production in general, it has to also serve as a starting point from which specific configurations of a measurement technique can be automatically generated. The latter ensures that the configuration of a measurement technique deployed to produce some measurement results adheres to the domain concepts of continuous security audits.

In order to develop such a language, first, common domain constructs which are needed to configure measurement techniques which are part of continuous security audit tools have to be identified. Thereupon, the language can be designed using a suitable formal grammar. Furthermore, to support configuration of specific measurement techniques, tool-specific code generators have to be developed which compile the target configuration language.

Ideally, the DSL should be developed in a way such that it can be used to configure any measurement technique which continuously produces evidence. In practice, however, this

requirement is hard to satisfy. The main reason for this is that measurement techniques supporting cloud service certification – and corresponding tool support – are subject to ongoing research and development where only a few prototypes are available as of writing this document. Furthermore, there are two fundamentally different approaches to continuous evidence production (6):

- **Monitoring-based evidence production**: These techniques use monitoring data as evidence which is produced during productive operation of a cloud-service (13). Two major types of monitoring-based evidence prodcution techniques can be distinguished: the first group consists of methods proposed by current research (e.g., Krotsiani et al. (10), Schiffmann et al. (14) which are specifically crafted to produce evidence to check whether particular properties of a cloud service are satisfied, e.g. integrity of cloud service components (14) and correctness of non-repudiation protocols used by cloud services (10)). Those methods require the implementation of additional monitoring services which are not needed for operational monitoring of the cloud service; the second group of monitoring-based evidence production techniques consists of existing monitoring services and tools which are used to operate the infrastructure of a cloud service, e.g. Nagios[1] or Ganglia[2]. The data produced by these monitoring tools can also be used as evidence to check a cloud service's properties such as availability (13). Also data produced by tools which aims to detect intrusions such as Snort[3], Bro[4], or OSSEC[5] can serve as evidence (13) (15).
- **Test-based evidence production:** Similar to monitoring-based techniques, test-based evidence production also collects evidence while a cloud-service is productively operating. Different to monitoring-based techniques, however, test-based techniques do not passively monitor operations of a cloud service but actively interact with it through tests. Thus test-based methods produce evidence by controlling some input to the cloud service, usually during productive operation, e.g. calling a cloud service's RESTful API (6) (16; 17).

It is reasonable to assume that configuring monitoring services underlying monitoring-based evidence production techniques and continuous test-based evidence production techniques will differ substantially. The work described in this document focuses on test-based measurement techniques, that is, measurement techniques which use test-based evidence

[1] https://www.nagios.org/
[2] http://ganglia.sourceforge.net/
[3] https://www.snort.org/
[4] https://www.bro.org/
[5] https://ossec.github.io/

production techniques to continuously produce evidence. Note that parts of the contents of this deliverable have been developed in (17).

1.2 WORKPACKAGE DEPENDIENCES

The unified configuration language introduced in this document has dependencies with Task 3.1 and 3.3 as shown in Figure 2. Consider Task 3.1 which defines data structures used to store instances of control objectives. Here, the unified configuration language allows to define candidate configurations, that is, templates for configurations of test-based measurement techniques and map them to corresponding control objectives to be continuously audited. Therefore, the output of this tasks serves as input to Task 3.1. In turn, fields of the control objective's data structure have to be available where to store the test-based measurement technique's configuration. That way, Task 3.2 also depends on input from Task 3.1.

Furthermore, the unified configuration language serves as input to Task 3.3 which defines a common data structure to represent evidence, i.e. instances of evidence produced by (test-based) measurement techniques. Unified configurations of test-based measurement techniques used to produce such evidence have to become part of the data structure of an evidence instance. Put differently: An instance of evidence has to contain the configuration of a test-based measurement technique which has been used to produce it. That way, Task 3.2 serves as input to Task 3.3.

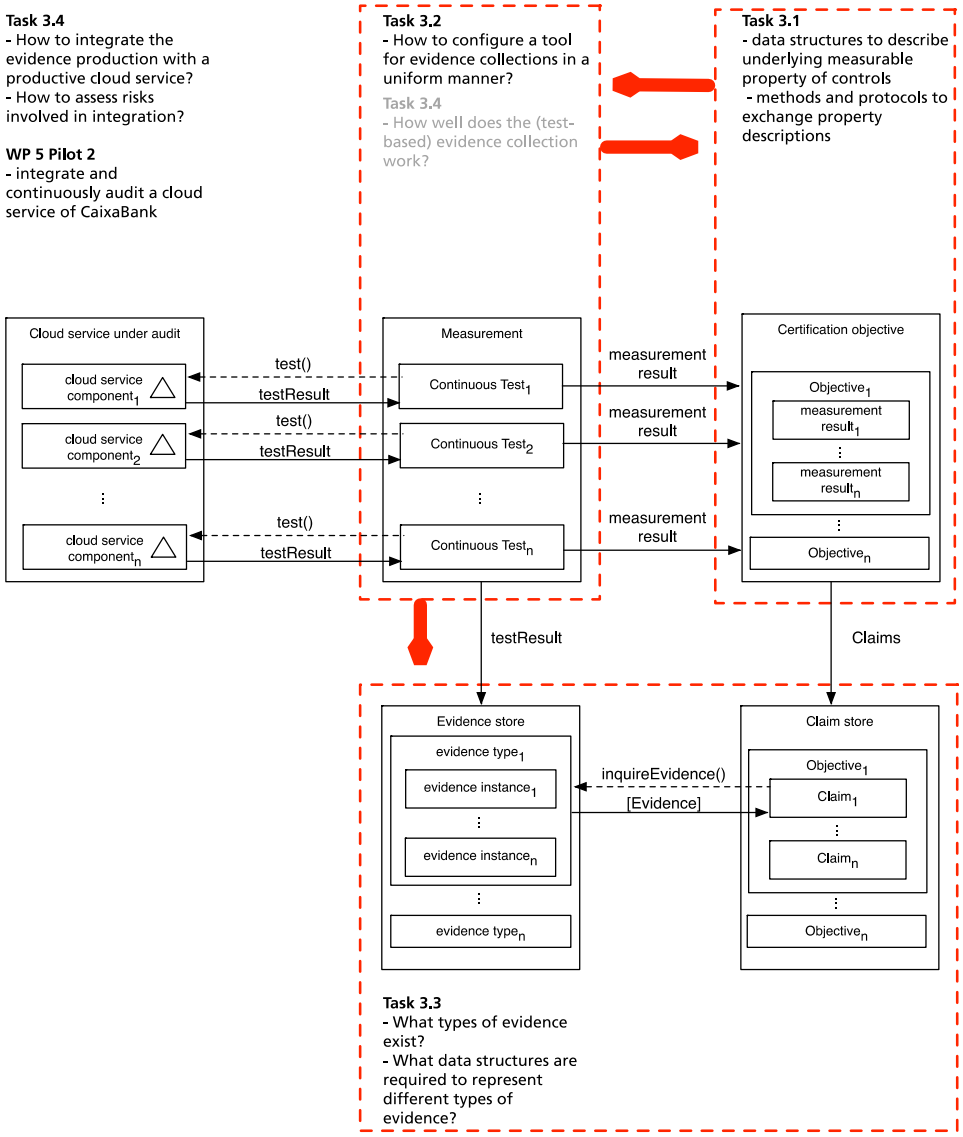


Figure 2: Dependencies of Task 3.2 within Working Package 3

1.3 ORGANISATION OF THIS DELIVERABLE

The remainder of this document is organized as follows: The next section outlines concepts which are needed in order to develop and define domain-specific languages. Thereafter, the DSL engineering process described by Mernik et al. (17) is followed (an outline of this process is described in Section 2.1): First, the decision to develop a DSL is explained, especially considering its contribution to Task 3.1 and Task 3.3 of Working Package 3. Thereafter, Section 4 describes the domain concepts of continuous test-based measurements (as part of continuous security audits), presents an exemplary implementation of these concepts which is called *Clouditor* as well as outlines three exemplary continuous test scenarios. Drawing on the

elicited domain concepts, Section 5 then identifies and scopes constructs which are required by a universal configuration language and, on this basis, describes the formal definition of the DSL using context-free grammars. Using this formal definition, Section 6 describes the implementation of the DSL using the language development tool XText[6] and shows how to use code generators to translate from the DSL to a target language, i.e. the configuration language of a specific measurement technique. Finally, Section 7 concludes this deliverable.

[6] https://eclipse.org/Xtext/

2 BACKGROUND

This section introduces basic concepts which are needed in order to develop and define domain-specific languages. The next section outlines necessary steps to develop a domain specific language which the remainder of this document will follow. Thereafter, the core concepts of formal languages are described while the focus lies on context-free grammars and their representation using Backus-Naur-Form (BNF) which is itself a domain specific language.

2.1 DSL ENGINEERING

There are multiple steps involved when developing a domain-specific language. In this section, we describe a process which was proposed by Mernik et al. (17) detailing all steps involved in DSL engineering. This process includes the steps *decision*, *analysis*, *design* and *implementation* of a DSL which are described hereafter.

- *Decision*: When developing a new DSL or identifying an existing DSL to reuse, the first step consists of properly motivating the usage of DSL because it initially incurs additional (often significant) effort. Such motivating can be driven by factors such as cost saving, e.g. a DSL helps eliminating repetitive and thus time-consuming tasks, or by correctness, e.g. facilitate the correct configuration of an application.

- *Analysis*: This step identifies, scopes and describes the domain for which the DSL is to be developed. To that end, different sources can serve as input to this analysis, including, for example, inspection of existing GPL code, technical documentation, and interview with domain experts. The necessary outcome of conducting this step is a description of the domain-specific terminology and semantics.

- *Design*: The design of a DSL can follow two main approaches: The first draws on an existing language where either some features of the existing language are reused (*piggyback*), restrict the existing language (*specialization*) or extend the existing language (*extension*). The second approach does not build on an existing language but aims at designing a DSL from scratch.

  Once it has been decided whether to invent a new language or to build on an existing one, the next step consists of defining the language either formally or informally (or both). Informal definition of DSL refers to using natural language to delineate the features of the DSL to be developed. Yet this approach is unsuited if the goal is to build a DSL that can actually be consumed by an application.

Therefore, we need to formally describe the syntax of the DSL which can be achieved using, e.g., EBNF as we will describe in Section 2.2.

- *Implementation*: Once a DSL has been formally defined, the language can be implemented. According to Mernik et al. (17), some exemplary choices include:
  - *Compilers* which translate the DSL constructs to constructs of an existing language and library calls (also known as *application generators*),
  - *interpreters* which recognize DSL constructs and interpret them,
  - *embedding* where DSL constructs, i.e. data types and operators are defined using constructs of an existing GPL, or
  - *compiler* or *Interpreter extensions* where the compiler or interpreter of an existing GPL is extended with code generation required for the DSL.

  Note that the implementation type *embedding* is also referred to as *internal* DSL. In this context, an *external* DSL is represented in a language different to main programming language it is interacting with (18).

  Implementing a DSL using a custom compiler or interpreter has many advantages, e.g., the syntax can be close to notations used by domain experts. However, it bears disadvantages such as having to implement custom, possibly complex language processors. Yet these disadvantages can be limited or eliminated if language development tools are used which automate most of the language processor construction (17). Examples for such tools are XText, Spoofax[7] or MPS[8].

After having outlined the main steps of the DSL engineering process, the following section introduces the necessary concepts of formal languages which are required to formally define the syntax of a DSL.

2.2 FORMAL LANGUAGES

A formal language L is defined by an alphabet $\sum$ and a grammar G. The alphabet $\sum$ is a set whose elements are called *symbols*. A finite sequence of symbols from $\sum$ are called *word*. The grammar G specifies which sequences of symbols are well-formed, that is, which word belongs to the language L.

A grammar G is defined by the 4-tuple (N,T,R,S) where

- N is the set of nonterminals, i.e. variables that represent language constructs,

[7] https://www.metaborg.org/
[8] https://www.jetbrains.com/mps/

- *T* is the set of terminals which is identical with the alphabet $\sum$ of the language *L*. Note that the set of nonterminals and terminals must not intersect, i.e. $N \cap T = \emptyset$.
- *R* is the set of productions which follow the form $l \rightarrow r$. Both l and r are sequences of nonterminals and terminals with l containing at least one nonterminal.
- *S* is a nonterminal (i.e., $S \epsilon N$) which constitutes the start variable.

2.2.1 CHOMSKY HIERACHY

Formal grammars can be classified according to the *Chomsky hierarchy* which distinguishes four types of grammars (19):

- *Type-0:* A grammar *G* which can be defined using the above 4-tuple is a grammar of type 0. A language generated by such a grammar is Turing-recognizable which means that a Turing machine exists which accepts all valid words of that language.
- *Type-1:* This type of grammar generates context-sensitive languages. The productions of these grammars may have more than one symbol on the left-hand side, that is, $|w_l| \geq 1$, provided that at least one of these symbol is a nonterminal. Thus the left-hand side may consist of terminals, i.e. symbols which are not replaced by the production, thereby establishing a context of the replacement. This is the reason why these grammars are called context-sensitive.
  Yet the size of the word on the left-hand side must not exceed the size of the word on the right-hand side of a production, that is, for all $|w_l| \rightarrow |w_r|$, the condition $|w_l| \leq |w_r|$ has to be satisfied. Furthermore, productions of the form $S \rightarrow \varepsilon$ are not permitted, except for *S* being the start symbol and not occurring on the right-hand side of any production. The latter two conditions implies that the size of a sequence as generated by a context-sensitive grammar always increases when a production is applied.
- *Type-2:* This type of grammar generates *context-free* languages. Opposed to context-sensitive grammars, the production of these grammars only allow the left-hand side to be nonterminal while words on the right hand-side may consist of both terminal and nonterminal symbols. Since no terminals are permitted on the left-hand side of a production, no context is considered during replacement. Hence, these grammars are context-free.
- *Type-3:* This type of grammar generates *regular* languages. The productions of this grammar restrict the left-hand side to single nonterminals while the right-hand side may either consist of a terminal followed by a nonterminal symbol (right regular) or a nonterminal followed by a terminal symbol (left regular).

2.2.2 EXTENDED BACKUS-NAUR FORM

The Backus-Naur Form (BNF) is a technique which supports the definition of define context-free grammars (20). Thus the BNF can be understood as a domain specific language itself which was developed with the purpose of easing syntax specification (17). Drawing on the BNF, the extended Backus-Naur Form (EBNF) (21) has been developed and standardized in ISO/IEC 14977:1996 Information technology - Syntactic metalanguage - Extended BNF (22).

Consider, as an example, the following context-free grammar G of the language $L = \{a^k b^k \mid k \geq 1\}$:

- $N = S$
- $T = \{a, b\}$
- $R = \{(S \rightarrow aSb), (S \rightarrow ab)\}$
- $S = S$

Deriving the word $w = aaabbb$ works as follows:

$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaabbb.$$

In order to use BNF to represent the grammar of this exemplary language, first some syntactic conventions have to be laid out: The symbol '::=' is used for productions instead of '→'. Also, the symbol '|' is used to represent alternative derivations more efficiently than stating alternative productions separately. Other variations include enclosing terminals in quotes to distinguish them from nonterminals which are enclosed with angle brackets, i.e '⟨ ⟩'. Using BNF, the context-free grammar to generate L can be described as

$$\langle S \rangle ::= 'a'S'b' \mid 'a''b'$$

The EBNF improves efficiency of defining context-free grammars further. Note that – despite the standardization effort in ISO/IEC 14977:1996 – there is no universally accepted variant of EBNF. Here, a version is chosen whose syntax is also heavily used by *xText*, an open source framework to implement domain specific languages, which will later be used to implement our test definition language (see Section 5.2). This EBNF version is also used to define, e.g., the formal grammar of XML (23). The most important extensions to syntactic conventions of BNF are the following:

- Nonterminal symbols are *not* enclosed with angle brackets because indicating terminal by single or double quotes is unambiguous,
- the '?' operator indicates that the symbol to the left is optional,
- the '*' operator defines that the symbol to the left can occur zero or multiple times, and

- the '+' operator defines that the symbol to the left occurs one or multiple times.

Using this variant of EBNF, we can define the grammar of language *L* as follows:

$$S ::= aS * b$$

3 DECISION TO DESIGN A UNIVERSAL TEST CONFIGURATION LANGUAGE

As already pointed out in the introduction, the motivation to develop a DSL which defines the configuration of test-based measurement techniques lies in having an approach at hand to formally define all required parts of a test-based measurement techniques in general. This not only paves the way for comparability of evidence and measurement results produced by these techniques but also guides the development of future test-based measurement techniques, thus having them conform with a common set of domain concepts defined for continuous security audits.

Since the DSL is defined by a formal grammar, the guarantee of a test-based measurement technique conforming with the domain concepts is *not* merely informal but is enforced through code generators: the developer has to provide a code generator which translates the constructs of the DSL into the target language constructs, that is, the language a specific tool uses to configure the test-based measurements. Put differently: any specific test-based measurement technique has to be configurable using a configuration written in the DSL. This implies that a suitable application generator exists which translates the constructs of the DSL into the language constructs which a particular test-based measurement techniques uses for configuration.

In context of Task 3.1 and 3.3 of Working Package 3, the DSL introduced in this document to formally define configurations of test-based measurement techniques depicts an essential contribution to ensure consistent semantics of evidence produced by these tools. The following sections outline the contribution the DSL makes to Task 3.1 as well as to Task 3.3.

3.1 TEST-BASED MEASUREMENT TECHNIQUE CONFIGURATION TEMPLATES (TASK 3.1)

Task 3.1 has control management at its heart, e.g. defining data structures used to store instances of control objectives to be audited continuously. The unified configuration language for test-based measurement techniques allows to define candidate configurations, that is, *templates* for configurations of test-based measurement techniques and map them to

corresponding control objectives. Those configuration templates are only partly defined and are complemented once the target of certification, that is, a cloud service to certify has been identified within a concrete scenario. For example, consider the exemplary case of checking the availability of a cloud service component described in the introduction of this document which used simple pings: configuring this test requires to, e.g., define the hostnames of the cloud service components whose availability should be tested as well as the expected round trip times. The continuous test template will only include placeholders for these parameters since they may change from one deployment of the continuous test to another one.

3.2 TEST-BASED MEASUREMENT TECHNIQUE CONFIGURATION EVIDENCE (TASK 3.3)

Task 3.3 of Working Package 3 centers around the question how to persist evidence which has been used to compute measurement results which, in turn, serve to check whether defined control objectives are satisfied. An integral part of Task 3.3 is therefore to define a common data structure to represent evidence, i.e. instances of evidence produced by (test-based) measurement techniques.

As already pointed out above, unified, formal definition of test-based measurement techniques' configurations allows to rigorously compare evidence produced by these techniques as part of the measurement. Consequently, configurations of test-based measurement techniques which are used to produce some evidence have to become part of the data structure of an evidence instance, that is, an instance of evidence has to contain the configuration of a test-based measurement technique which has been used to produce it. Drawing on the DSL, the means to represent the configuration of a test-based measurement technique in a general manner are available which can be included as part of the evidence data structure.

4 CONTINUOUS TEST-BASED MEASUREMENTS

In this section, domain specific constructs are described which are used by test-based measurement techniques which are part of continuous security audits supporting continuous security certification of cloud services (see Figure 1). To that end, the next section introduces the building blocks of continuous test-based measurement techniques. Thereafter, Section 4.2 outlines *Clouditor* which delineates one exemplary implementation of the building blocks. Finally, Section 4.3 presents three exemplary scenarios of continuous, test-based security audits of cloud service components.

4.1 BUILDING BLOCKS OF CONTINUOUS TEST-BASED MEASUREMENT TECHNIQUES

This section introduces the five main building blocks of continuous test-based measurement techniques. First, an overview of the core concepts are provided, outlining how they can be used to design a continuous test-based measurement (Section 4.1.1). Thereafter, each building block is explained in detail (Section 4.1.2 – 4.1.6).

4.1.1 OVERVIEW

Continuous test-based measurement techniques use continuous tests to automatically and repeatedly reason about security properties of cloud services. A continuous test CT consists of five building blocks: *Test suites* (TS) define any single test which is executed repeatedly within a continuous test. When defining a test suite, one or more *test cases* (TC) are associated with the suite. A *test case* forms the primitive of any continuous test, it specifies the concrete steps to test a cloud service as well as to evaluate whether a test case passed or failed. Test cases do not depend on specific test suites and can therefore be reused with any suite if needed. The result of a test suite - in its simplest form passed or failed - are used in two ways: on the one hand, test suite results are used to compute *test metrics* (M) which allows to compute measurement results to evaluate statements over a cloud service's property (i.e., control objectives), e.g. a detected security vulnerability of a cloud service was fixed within 24 hours. On the other hand, test suite results are used by the *workflow* (W) to decide which test suite to execute next. Lastly, we have to test whether the assumptions made about the environment of a cloud service under test hold which we refer to as testing *preconditions* (TP).

Figure 3 shows how the building blocks constitute a continuous test: initially, the workflow decides which test suite to run first (Step 1). Executing a test suite translates to executing any test case contained within the suite. The result of the test suite is then supplied to one or more test metrics (Step 2a). Furthermore, the test suite result is handed over to the workflow (Step 2b) which, based on the result, decides which test suite to execute next (Step 3). Upon completion, the results of the test suite are again used to compute test metrics (Step 4a) and supplied to the workflow (Step 4b) deciding which test suite to execute next and so forth.

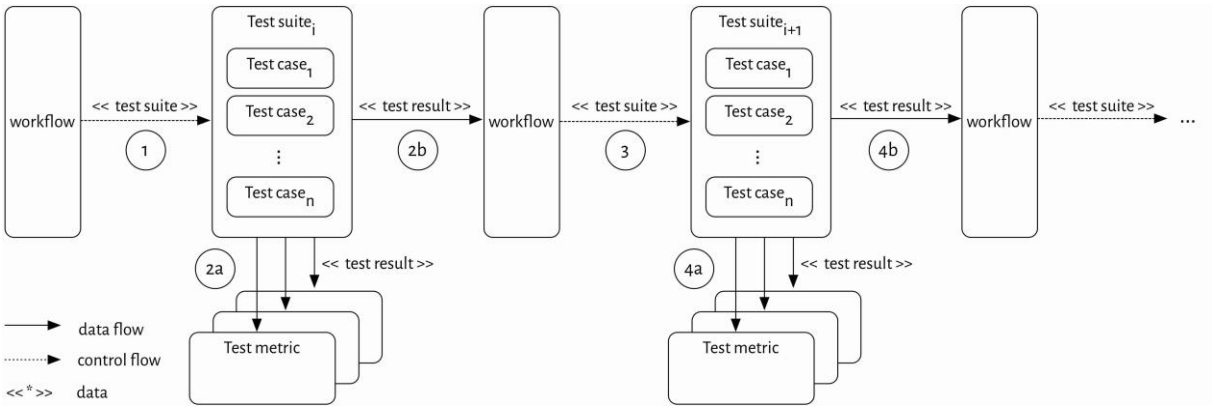


Figure 3: Overview of building blocks constituting continuous test-based measurements

4.1.2 TEST CASES

Test cases are the primitive of any continuous test. Each test case consists of *procedures* which specify any steps that are executed by the test case. For example, a test case may specify to establish a SSH connection to a virtual machine (VM), then issue a command to download and install a package on the machine. In order to execute correctly, a procedure may require *input parameters*, e.g. successfully connecting to a VM via SSH requires *username*, *hostname*, and *path_to_private_key_file*. The arguments which are passed to a procedure's input parameters can be selected randomly from a predefined set, e.g. which application to download and install on the VM is selected randomly from the package list.

Further, each test case has a set of *oracles*, that is, methods which are used to determine whether the results of a test case indicates failure or success. In order for a test case to pass, all defined oracles have to indicate success. Yet besides simply passing or failing, the result of a test case also includes start and finishing time of the test case, i.e. time elapsed between starting a test case run and completing reasoning about the test results. Also, the test case result can provide further information, for example, the maximum average response time of TCP packets measured to test latency of the connection to a remote host.

Lastly, a test case possesses an *ordering number* which serves to specify the priority with which a test case is executed as part of the test suite (see next section). Consider, for example, a test suite which has three test cases TC_1, TC_2 and TC_3 where TC_1 and TC_2 have ordering numbers 1 and 1, and TC_3 has ordering number 2. When this test suite is executed, then TC_1 and TC_2 will be executed firstly and concurrently. As soon as both TC_1 and TC_2 have completed execution, execution of TC_3 is triggered.

More formally, we can describe a test case TC as the 4-tuple which consists of the following four elements: Procedures E where each procedure $e \in E$ requires a tuple of input parameters $P = \langle p_1, p_2, \dots, p_i \rangle \in L$. L is the ordered list which contains any input parameter tuples required for the defined procedures of a test case TC. Furthermore, TC consists of a tuple of oracles O where each oracle $o \in O$ evaluates if a test case passed or failed, as well as an ordering number $N \in \mathbb{N}^+$:

$$TC = \langle E, L, O, N \rangle.$$

Recall the exemplary test case of connecting to a VM and installing a package: This test case may contain the three procedures

$$E = \langle connect\_via\_ssh, install\_package, compute\_mac \rangle,$$

where a SSH connection requires input parameters

$$P\_1 = \langle username, hostname, path\_to\_private\_keyfile \rangle,$$

installing a package using *apt-get install* requires input parameters

$$P\_2 = \langle package\_name \rangle,$$

and, finally, computing a message authentication code (MAC) of the installed package using openssl dgst -sha256 -hmac requires input parameters

$$P\_3 = \langle key \rangle.$$

Furthermore, the test case passes if the MAC of the installed package and a MAC which was previously computed and stored by the oracle match:

$$O = \langle compare\_mac \rangle.$$

Finally, the test case executes immediately when the test suite execution is triggered, that is, its ordering number is $N = 1$. In summary, we can describe the exemplary trusted package installation (TPA) test case as follows:

$$TC^{TPI} = \langle\langle connect\_via\_ssh, install\_package, compute\_mac\rangle,$$
$$\langle\langle username, hostname, path\_to\_private\_key file\rangle, \langle package\_name\rangle, \langle key\rangle\rangle,$$
$$\langle compare\_mac\rangle,$$
$$1\rangle.$$

As mentioned above, arguments passed to an input parameter can be randomized. As an example, consider the input parameters $p_{21} = package\_name$ and $p_{31} = key$ where the package to be installed as well as the key used for computing the MAC can be selected randomly. We describe a random argument with values in V as a function $A: \Omega \rightarrow V$ where Ω is the set of all possible arguments that can be passed to an input parameter $p \in P$. In our example, Ω of p_{21} contains all valid package names while A can evaluate to, e.g., $mysql\_server$.

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

4.1.3 TEST SUITES

A test suite combines test cases where each suite contains at least one test case. Hereafter, we refer to the execution of a test suite as test suite run (tsr). Once the test suite run completes, it returns failure or success. A test suite either passes or fails, it passes if all contained test cases pass. Furthermore, upon completion, the test suite run returns the start (tsr^s) and end time (tsr^e), as well as the results of all bound test cases.

A test suite can be executed *successively* multiple times which is defined by *iterations*, e.g. 100. Triggering execution of a test suite translates to triggering execution of test cases bound to the test suite. Test cases with smallest ordering number are executed first and, having returned, are followed by test cases with next larger ordering number. In order for the following test suite's iteration to start, the current iteration of a test suite has to be completed, that is, any test cases bound to the test suite have to be completed. The number of successive iterations can be set to infinity. In this case, consecutively triggering a test suite's execution will not terminate until otherwise interrupted, e.g. by a decision made by the workflow (see following section).

A test suite also defines an *interval* which describes the period of time in seconds between consecutive executions of a test suite. One option to configure the interval is to trigger execution of a test suite after a fixed interval passed, e.g. 600 seconds after the previous test

suite execution completed. Alternatively, the interval can serve as a window from which the start of a test suite's execution is selected randomly. A special case consists of individually fixed intervals per iteration: Here, each interval prior to execution of a test suite is fixed – and thus not chosen randomly – but assumes an individual value. For example: a test suite is configured to run three times successively, i.e. the number of iterations is three, where each waiting interval before executing the test suite is defined individually, e.g. wait 2 seconds before the first iteration, 4 before the second iteration and 8 before the third iteration.

Lastly, if subsequent iterations of a test suite start instantaneously, then they may produce unwanted side effects. In order to prevent such correlations, a fixed *offset* (seconds) can be defined permitting the service instance under test to clean up after a test suite has completed.

A test suite TS is described as the 4-tuple which consists of the following four elements: Bound test cases $\mathcal{TC} = \langle TC^1, TC^2, \ldots, TC^n \rangle$, the number of iteration $I \in \mathbb{N}^+$, the offset $F \in \mathbb{N}^+$ (seconds), between test suite executions as well as the interval $T \in \mathbb{N}^+$ (seconds) which specifies either the fixed or randomized time between consecutive test suite iterations:

$$TS = \langle \mathcal{TC}, I, F, T \rangle.$$

To illustrate the usage of a test suite, recall the trusted package installation test case TC^{TPA} described in the previous section. As an example, lets assume that the execution of TC^{PI} is triggered randomly within a time interval of 60 minutes, i.e. $T = 3600$. Furthermore, the test suite is consecutively executed for 3000 times, i.e. $I = 3000$, having a 15 minute offset between every execution, that is, $F = 900$. Consequently, the exemplary test suite containing a single test case TC^{TPI} can be described as follows:

$$TS^{\langle TPI \rangle} = \langle \langle TC^{TPI} \rangle, 3000, 900, [0, 3600] \rangle.$$

4.1.4 WORKFLOW

A workflow defines dependencies between iterations of a test suite and between iterations of different test suites. To that end, a workflow uses the results of test suites to control the execution of other test suites. Consider, as a basic example, that after successfully completing a number of iterations, a test suite fails. A workflow can now define what to do as a reaction to this failure, e.g. whether to continue running the test suite for the remaining iterations, to start

another test suite or terminate the test. Therefore, a workflow permits fine-grained control of the execution flow of a continuous test.

Recall the exemplary test suite $TS^{\langle TPI \rangle}$ which contains the test case TC^{TPI}: if $TS^{\langle TPI \rangle}$ fails at the third iteration, i.e. $I = 3$, then the workflow may stop execution of $TS^{\langle TPI \rangle}$ and trigger a different test suite which checks integrity of packages previously installed on the VM to determine whether their integrity has also been compromised.

A workflow can be described as the function $\mathcal{W} : R \rightarrow T\,S$ which takes as input the results of executed test suites R where each $r \in R$ is a 2-tuple of a test suite $TS^{\langle TC \rangle}$ and a sequence of test suite's results Y after the i-th iteration, that is, $|Y| = i$. For example, the input for $\mathcal{W}$ for test suite $TS^{\langle TPI \rangle}$ after the third iteration is $r = \langle TS^{\langle TPI \rangle}, \langle passed, passed, failed \rangle\rangle$. For each test suite's results, W outputs the test suite $T\,S \in \mathcal{TS}$ to be executed next, for example, to trigger execution of a different test suite if the current test suite failed for the last five consecutive iterations.

4.1.5 TEST METRICS

Continuous test-based measurement techniques aim at automatically and repeatedly produce measurement results which are used to check if a cloud service complies with a set of control objectives over time. Thus it is necessary to interpret a sequence of test results in order to reason about cloud service properties over a period of time. To that end, suitable metrics are computed which permit us to evaluate statements over cloud services properties such as the availability of the cloud service is higher than 99.999% per day.

The computation of a metric $m \in M$ can be described as the function $CM : R \rightarrow M$ which takes as input results of test suite runs R. 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.

Note that singular test (suite) results already denote the most basic type of a test-based measurement results. This means that only parts of a singular test result can be considered *evidence* whereas the test result already implies that a decision has been made based on the information obtained during the test's execution. More specifically, any information which serves as input to a well-defined test oracle which are part of test cases constitutes *evidence*. Therefore, the test result itself has to be considered a measurement result since the test oracle delineates the primitive of a test metric. However, only considering a singular test result does

not allow to reason about periods of time, hence test metrics composed of multiple test results are required.

4.1.6 PRECONDITIONS

Naively executing continuous tests is prone to produce false negative test results, e.g., testing if a webserver's TLS configuration is secure may fail not because of a vulnerable configuration but because the webserver cannot be reached. Computing test metrics based on false negative test will lead to erroneous metrics and thus incorrect evaluations of statement over the cloud services. Therefore, the assumptions made about the environment of the cloud service under test, i.e. preconditions, need also to be tested.

There are two options to model preconditions using the building blocks of continuous tests. These options are explained in the following two paragraphs.

Precondition as specialized test suites This option treats preconditions as a special type of test suite: First, test cases are design which aim to check whether preconditions hold. A specialized test suite is then created which only bind these precondition test cases. Finally, this specialized test suite has to be executed prior to the *main test suite*, i.e. the test suite designed to reason about a cloud service's property. To that end, a workflow is defined which only executes the main test suite if all preconditions have passed. Therefore, preconditions can be used to control the workflow of a continuous test, allowing to adapt, i.e. select and execute test suite according to environmental conditions discovered at runtime.

Figure 4 shows an extract of an exemplary continuous test that use specialized test suites to test preconditions before executing the main test suite: after having successfully tested the preconditions (Step 1), the workflow triggers execution of the main test suite (Step 2). After having executed the main test suite, the test result is used to compute test metrics (Step 3a)

and supplied to the workflow (Step 3b) which triggers execution of specialized test suite to again validate the preconditions (Step 4) and so forth.

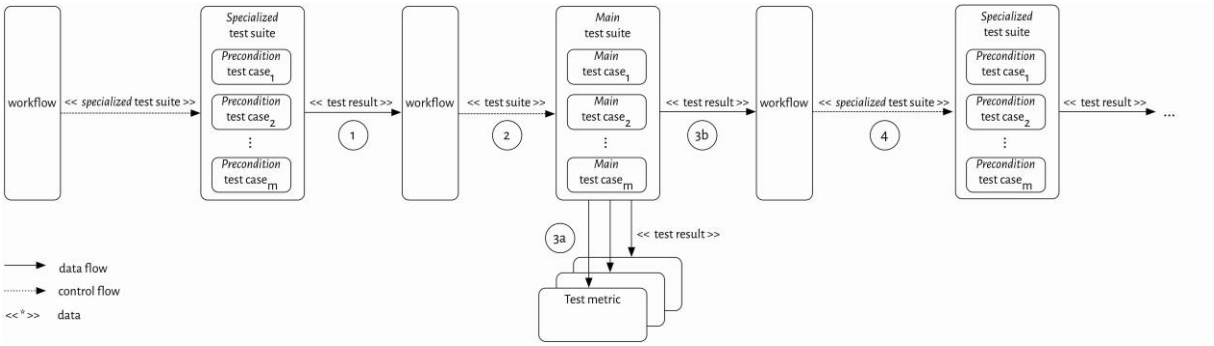


Figure 4: Extract of an exemplary continuous test using specialized test suites to test preconditions before executing main test suites

Consider, as an example, that a continuous test aims to check whether the bandwidth available to a VM for uploads is at least 50 Mbit per second. To that end, first a connection to the VM via SSH is established, then a file is uploaded where measuring the duration of that upload. One exemplary precondition for this test to execute correctly is that the VM is reachable via SSH. In order to evaluate whether this precondition holds, it is possible to probe the VM's port 22 by sending a SYN TCP segment and check if the host response with a SYN-ACK segment. Only if the precondition test suite determines that a VM's port 22 is accessible, then the bandwidth test is executed.

Using specialized test suites to model preconditions has one important drawback: As described in Section 4.1.3, test suites are executed successively, that is, execution of the next suite is triggered once the previous suite completed execution. Thus, a test suite containing preconditions may have passed but during the following main test suite, the preconditions are not satisfied anymore. Consequently, the main test suite may incorrectly fail, producing an inaccurate test result.

Preconditions as part of main test suites The second option consists of modeling pre-conditions as test cases and binding them to the main test suites. Figure 5 shows that after the workflow triggered execution of the test suite (Step 1), these precondition test cases are executed concurrently with the main test cases. Since a test suite only passes if all contained test cases pass (see Section 4.1.3), a failing precondition test case leads to a failing test suite. In order not to misinterpret a failed test and thus create a false negative test result, a failed test

suite result is inspected during test metrics' computation (Step 2a). If any precondition test case failed, then test result is ignored during computation of metrics.

Having preconditions as part of main test suite also allows to control the workflow. After having provided the test suite result to the workflow (Step 2b), the workflow inspects the test suite results and selects the next test suite to execute accordingly (Step 3). However, concurrently executing precondition test cases and main test cases comes at a price: regardless of any precondition test case failing, the remaining precondition test cases as well as the main test cases of the test suite are still executed, although the result of the test suite will be discarded.

Modelling preconditions as part of main test suites has the advantage that it allows to enforce that testing preconditions and the main test suite are executed concurrently. Moreover, note that the ordering numbers – which are required for definition of test cases (see Section 4.1.2) – allow for fine-grained pairing of precondition test cases and main test cases within the test suite.

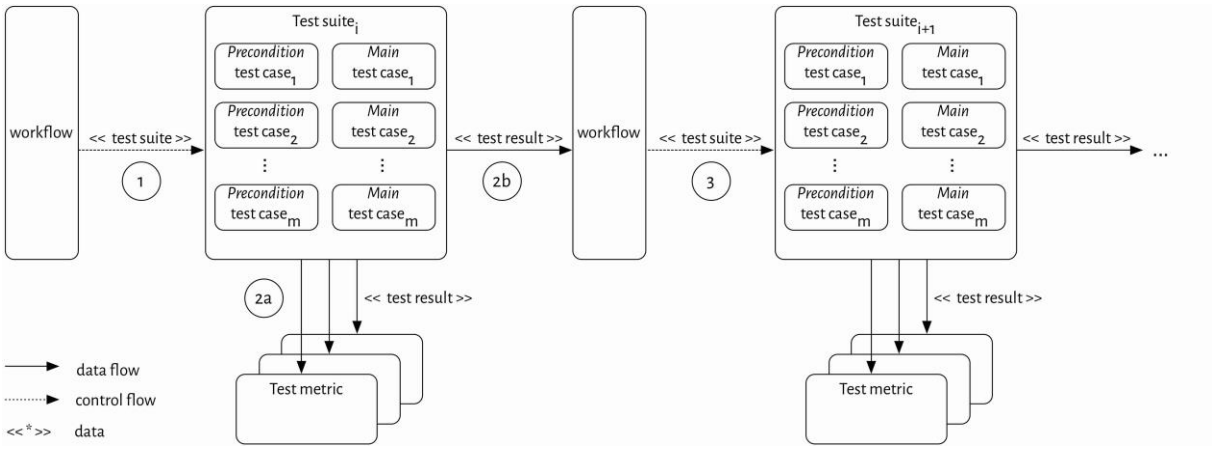


Figure 5: Extract of an exemplary continuous test using precondition test cases as part of the main test suites

4.2 CLOUDITOR: AN EXEMPLARY TOOL TO IMPLEMENT TEST-BASED MEASUREMENT TECHNIQUES

This section first outlines the Clouditor-engine which is part of the Clouditor toolbox and whose design follows the building blocks described in the previous section. The Clouditor toolbox is part of the background of the EU-SEC project, it consists of five main components – Engine,

Explorer, Simulator, Evaluator & Dashboard – which are shown in Figure 6[9]. The engine is responsible for executing test-based measurement techniques. To that end, it implements and deploys continuous tests following the building blocks described in the previous section.

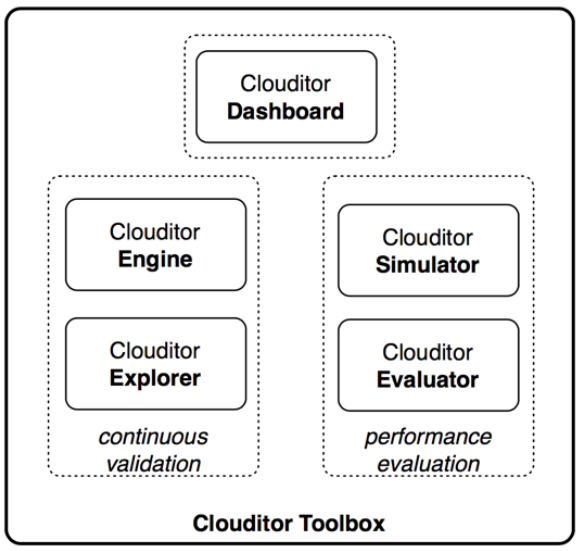


Figure 6: Components of the Clouditor toolbox

Figure 7 shows a high level architecture of the Clouditor Engine's components, including data and control flow. Test case are implemented using hooks to existing security tools such as Nmap[10], SQLMap[11], sslyze[12] etc, there reusing existing knowledge and tooling. Alternatively, test cases can be implemented natively and self-contained as part of the Engine.

[9] For further information about the Clouditor toolbox see https://www.aisec.fraunhofer.de/de/fields-of-expertise/projekte/Clouditor.html.
[10] https://nmap.org/
[11] http://sqlmap.org/
[12] https://github.com/nabla-c0d3/sslyze

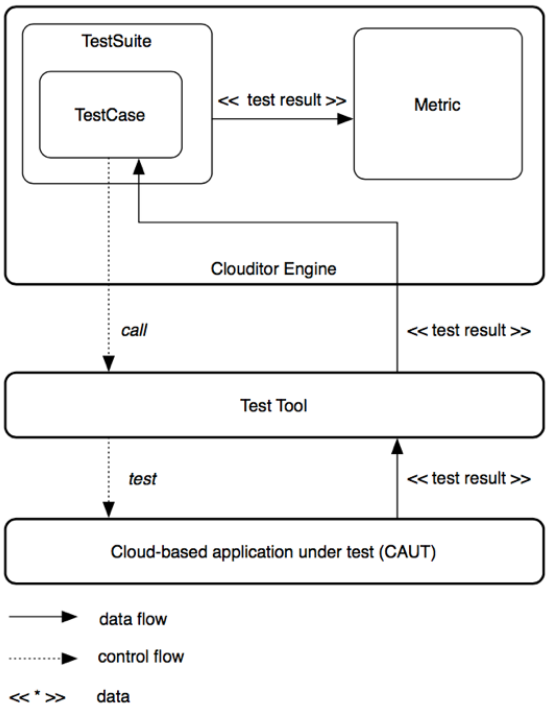


Figure 7: Overview of Clouditor Engine main components (with external test tool)

4.3 EXEMPLARY CONTINUOUS TEST SCENARIOS

This section presents three exemplary scenarios which are implemented using the Clouditor-Engine. Each of the exemplary scenarios outlines the property of the cloud service that the test is aiming at, an exemplary configuration of the test-based measurement technique as well as candidate controls for which the measurement techniques can provide measurement results.

4.3.1 CONTINUOUSLY TESTING SECURE COMMUNICATION CONFIGURATION

As the name suggests, secure communication configuration is a type of security property of a cloud service which holds if communication with the cloud service is secure against disclosure and manipulation by unauthorized parties. Since customer usually accesses cloud services remotely using insecure networks, securing communication end-to-end, that is, between the service and the customer is an indispensable necessity.

Protocols used to securely communicate with cloud service endpoints vary depending on the type of cloud service, i.e. the cloud service model. While securely communicating with IaaS

translates to, e.g., using SSH to connect to a virtual machine, secure communication with PaaS and SaaS applications may use HTTPS. In the latter case, HTTP uses Transport Layer Security (TLS) where TLS is a widespread cryptographic protocol aiming to secure communication over untrusted networks. However, configuring TLS properly is not trivial because it supports various methods for key exchange, encryption, and authentication (24). A concrete set of such methods used to secure communication is referred to as a cipher suite where some cipher suites are considered insufficient to provide secure communication, e.g. if they use the stream cipher RC4 as an encryption algorithm (25).

The continuous test *TLSConTest* can be used to continuously check whether securely communicating with the endpoint of a cloud service is feasible. This test continuously evaluates if the SSL/TLS configuration of a cloud service's web server allows to securely communicate with the service.

An exemplary configuration of *TLSConTest* can be defined as follows: every ten seconds, first preconditions are tested which establish that the endpoint of the cloud services is reachable via ICMP and TCP. In case these preconditions are satisfied, following the precondition test, the SSL/TLS configuration of the endpoint is tested. It fails if the cloud service endpoint exhibits, e.g. a known SSL/TLS vulnerability, uses self-signed certificates or supports vulnerable cipher suites. Further, the option *Precondition as specialized test suites* introduced in Section 4.1.6 is used. This means that if one or both precondition test cases fail, then testing of the SSL/TLS configuration is not executed. In this case, preconditions are tested again in the following iteration, i.e. ten seconds after the previous precondition test have completed. Finally, the results produced by *TLSConTest* indicate how often the SSL/TLS configuration of the cloud service under test is insecure and how long it takes to fix these misconfigurations.

Measurement results produced by *TLSConTest* can support continuous security audits according to the following controls, e.g.

- KRY-02 Encryption of data for transmission (transport encryption) of BSI C5 (1),
- EKM-03: Encryption & Key Management Sensitive Data Protection of CSA's Cloud Controls Matrix (CCM) (26) as well as
- A.14.1.2 Securing application services on public networks of ISO/IEC 27001:2013 (27).

4.3.2 CONTINUOUSLY TESTING INPUT VALIDATION

Software-as-a-Service (SaaS) are applications which are deployed on remote infrastructures and which are usually accessible through interfaces such as browsers or standalone program

interfaces. The control of the customer over the application is usually confined to configurations of user-specific application settings (28). Providing as well as using SaaS or SaaS-based applications thus requires comprehensively employing web application technologies, e.g. JavaScript, JSON, HTML and CSS. As a result, SaaS inherits potential web application vulnerabilities, for example, they can be vulnerable to SQL injections or session hijacking (29).

The Open Web Application Security Project (OWASP) defines a list of ten categories of web application vulnerabilities which are supposed to contain the most frequently found vulnerabilities in the wild (30). The category *A1 - Injection* leads that list, thus making it the most prevalent type of web application vulnerability. While Injection covers various types of vulnerabilities, e.g., SQL, OS commands and LDAP injection, SQL injection (SQLI) is among the most common types of vulnerabilities which web applications possess. If a web application is vulnerable to SQLI, then malicious code can be inserted into query strings which are parsed and executed by the SQL server, potentially leading to, e.g., disclosure of confidential data stored in SQL database or bypassing user authentication (31).

Consider, as an example, that at some point in time auditing a SaaS application reveals that it possesses SQL injection vulnerabilities. Assuming that, as a reaction, data sanitization is implemented at the database layer using stored procedures which depicts one possible countermeasure. However, if this exemplary SaaS application makes use of framework such as Ruby on Rails[13], then changing the database used by the application's controller is achieved through simple configuration changes. In case the newly deployed database instance does not use the previously introduced stored procedures to sanitize user input, then previously fixed SQLI vulnerabilities are reintroduced. Further, a SaaS provider does not need to possess the resources which are used to create and deploy the web application components but may leverage a Platform-as-a-Service (PaaS) provider such as Google App Engine[14]. As a result, another layer of abstraction is added to the architecture of the SaaS application where changes in the backend rendering the SaaS application vulnerable are hard to detect, even for the SaaS provider herself.

Checking SQLI vulnerabilities of SaaS application thus requires an approach capable of continuously, i.e. automatically and repeatedly check whether the cloud service validates user input. To that end, the continuous test *SQLContTest* can be used which continuously tests web application components of a SaaS application for SQLI vulnerabilities.

[13] https://rubyonrails.org/
[14] https://cloud.google.com/appengine/

Consider the following exemplary configuration of *SQLContTest*: every 30 seconds, it uses a URL of the cloud service under test to scans this endpoint for SQLI vulnerabilities. If any vulnerabilities are indicated by the scan, then the test fails, otherwise it passes. The test results produced by the continuous test aim at counting the times during which the cloud service is vulnerable to SQLI within a particular period of time.

Measurement results produced by this continuous test can, for example, support continuous security audits according to the following controls:

- RB-21: Handling of vulnerabilities, malfunctions and errors – check of open vulnerabilities of BSI C5 (1),
- TVM-02: Threat and Vulnerability Management Vulnerability / Patch Management of CSA's CCM (26) as well as
- A.12.6.1 Management of technical vulnerabilities of ISO/IEC 27001:2013 (27).

4.3.3 CONTINUOUSLY TESTING SECURE INTERFACE CONFIGURATION

Secure interface configuration is another security properties of a cloud service which is satisfied if a cloud service component only exposes those interfaces publicly which are actually intended to be publicly reachable. Common configuration flaws can render a cloud service vulnerable which, in case of an attacker manages to exploit this vulnerability, can lead to, e.g., disclosure or manipulation of valuable data stored and processed by the cloud service.

Consider, for example, the Amazon Relational Database Service (AWS RDS)[15], a PaaS application which provides industry-standard relational database as a web service. This application uses a special type of security groups, called Amazon RDS Security Groups[16]. These security groups are used to control what IP addresses or other Amazon resources such as EC2 instances have access to the database service instance. Erroneous configurations of these security groups may expose the database service to unauthorized access.

The continuous test *PortConTest* is proposed to determine whether a cloud service temporarily exposes interface due to insecure configurations. *PortConTest* continuously probes the endpoint of a cloud service, either an hostname or IP address, for open ports which should not be publicly accessible.

An exemplary configuration of this continuous test can be summarized as follows: *PortConTest* tests every 30 seconds if the endpoint of the cloud service under test can be reached via ICMP

[15] https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide//Welcome.html
[16] https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide//Overview.RDSSecurityGroups.html

and, at the same time, probes the endpoint for open ports. Testing the reachability of the target host on the Internet Layer is a precondition test case which is executed concurrently with the test for open ports. Thus *PortConTest* makes use of the preconditions as part of main test suites of our framework described in Section 4.1.6. As a result, the result of the port scan is only considered if the precondition holds, i.e. if the target host can be reached via ICMP. The results produced by *PortConTest* show how long it takes the cloud service provider to fix in secure interface configurations of the cloud service under test.

Measurement results produced by *PortConTest* test can, e.g., support continuous security audits according to the following controls:

- RB-22 Handling of vulnerabilities, malfunctions and errors – system hardening of the BSI C5 (1),
- IVS-06: Infrastructure & Virtualization Security Network Security of CSA's CCM (26) as well as
- A.9.1.2 Access to networks and network services of ISO 27001:2013 (27).

5 DESIGN OF A UNIVERSAL CONFIGURATION LANGUAGE

This chapter introduces the design of the universal configuration language called *ConTest*. The goal of ConTest is to strictly define the configuration of test-based measurement techniques in a general manner. The following section identifies and scopes required language constructs. This analysis draws on the domain specific constructs used by continuous, test-based measurements as part of security audits which were introduced in Section 4.1. Thereafter, the context-free grammar which generates ConTest is defined (Section 5.2).

5.1 IDENTIFICATION AND SCOPING OF REQUIRED LANGUAGE CONSTRUCTS

This section identifies and defines the scope of the required constructs which the DSL *ConTest* has to provide. To that end, the description of the building blocks presented in the previous section are used.

5.1.1 TEST CASE

Recall that a test case TC consists of four elements: *Procedures* (E), an ordered List of *input parameters* (L), an *oracle* (O) and an *ordering number* (N):

$$TC = \langle E, L, O, N \rangle.$$

As the name implies, *procedures* describe the actual steps taken during a test case. Including such procedural details in the universal configuration language is unnecessary because the implementation of the procedures is left the developer implementing the test-based measurement technique. Rather, all procedures of a test case are summarized by a construct

named *TestCaseModule* which points to a particular component of the measurement technique which implements the necessary procedures.

The *list of input parameters* which are used as input to the procedures are included in the configuration language. An implementation of a test-based measurement technique can then pass the values specified for input parameter to the *TestCaseModule*.

Test oracles are mechanisms to decide whether a test passes or fails. Similar to the case of procedures, a procedural description of the oracle is not included in ConTest given how the actual evaluation of test case results is left to any concrete implementation. Yet defining parameters which the oracle uses to reason about test results, e.g. in form of Boolean expressions, is needed. To that end, a list of assert parameters which a continuous test implementation passes to the test oracle is included in ConTest. The *ordering number* is used to prioritize test cases' execution as part of a test suite. The ordering number is included in ConTest. Finally, each instance of a test case that is specified as part of a measurement technique's configuration has to be addressable through a unique ID (unique in scope of the test configuration instance).

5.1.2 TEST SUITE

A set of test cases $\mathcal{TC}$ containing one or more test cases TC are combined to a test suite which also consists of the number of iterations (I), an offset (N) and the interval (T):

$$TS = \langle \mathcal{TC}, I, F, T \rangle.$$

Test cases which are part of a test suite have to be included in the measurement technique's configuration. It was described in the previous section which part of a test case has to be represented by ConTest. In order to bind a test case to a test suite within a measurement technique's configuration, the unique ID of a test is used.

The *iterations* of a test suite, that is, how many times the test suite is to be executed during a continuous test as well as the *offset*, i.e. the fixed waiting time between two test suite executions are included in the configuration of the test-based measurement technique. Also the interval between two text suite executions is specified as part of the configuration where it is important

to support either specifying a fixed interval, the range from which a random value for the interval is selected or individually fixed intervals per iteration.

Finally, and similarly to the test case definition, each instance of a test suite requires a unique ID in scope of an instance of the test-based measurement configuration.

5.1.3 WORKFLOW

Deciding what test suite to execute next is the responsibility of the workflow. A test-based measurement technique uses exactly one workflow.

The configuration of a test-based measurement technique does not include the procedural elements of a workflow but only a pointer to the *WorkflowModule* which a measurement technique implementation uses. Furthermore, the configuration has to include the test suites which a workflow may use. To that end, defined test suites are bound to the workflow using their unique ID.

5.1.4 TEST METRICS

Test metrics allow to reason about a sequence of test suite results produced by a test-based measurement technique. A test-based technique may compute one or more test metrics. The actual procedures which a test metric may use are not included in the configuration of the measurement technique because those are specific to the implementation. Similar to a workflow definition and test case definitions, defining test metrics includes a pointer to *TestMetricModule*, i.e. the part of the implementation of the test-based technique where the test metric is actually computed.

5.1.5 PRECONDITIONS

In order to test assumptions made about the environment of the cloud service under test, preconditions are used. Since preconditions can be either designed as a specialized test suite or as precondition test cases (for further details see Section 4.1.6), no additional constructs for the universal configuration language are needed.

5.2 FORMAL DEFINITION OF CONTEST

This section uses Extended-Backus-Naur Form (EBNF) to define the context-free grammar which generates ConTest (Figure 8). Terminal symbols are bold to improve readability of the grammar. Hereafter, the grammar is explained line by line, thereby relating the designed constructs to the analysis conducted in the previous section.

The start symbol of ConTest's grammar is ConTest from which the first rule derives the variable Test. The rest of the grammar of ConTest is built as follows:

- Lines 3 to 9: *Test* is defined by the '*TestID*' which is followed by the variable *ID* which assigns a unique Id to a configuration of a measurement techniques. Further, '*TestName*' is followed by the variable *String* providing a name with a descriptive name. Further, *Test* is defined by exactly one *Workflow* and by one or more *TestMetric*. Definitions of these two variable are provided in the following two paragraphs.
- Lines 11 to 16: *TestMetric* is defined by a unique '*TestMetricID*' which is followed by the variable ID. Also, *TestMetric* is defined by the terminals '*TestMetricName*', '*TestMetricModule*', and '*Description*' each of which – while grouped with curly braces '{' and '}' for better comprehensibility – is followed by the variable String. Whereas '*TestMetricName*' and '*Description*' are self-explanatory, the *String* following '*TestMetricModule*' specifies the component of a concrete test-based measurement technique which implements the desired test metric, e.g. a particular class or module.
- Lines 18 to 22: Similar to *TestMetric*, the variable Workflow is defined by the terminals '*WorkflowID*' followed by *ID*, as well as '*WorkflowName*', and '*WorkflowModule*' which are each followed by the variable String. Similar to '*TestMetricModule*', the '*WorkflowModule*' defines the component of a specific continuous test implementation. Further, *Workflow* is defined by one or more *TestSuite* which are enclosed by curly brackets for better readability.
- Lines 24 to 37: *TestSuite* is defined by the terminals 'TestSuiteID' followed by *ID* and 'TestSuiteName' followed by the variable *String*. Also, *TestSuite* is specified through '*NumberOfMaxIteration*' followed by the variable *Int* which specifies the upper bound of iterations a particular test suite is executed during a continuous test-based measurement.

  Next, there is the terminal '*IntervalBetweenTests*' which is followed by either the terminal '*fixedInterval*' with variable *Int*, by '*randomizedInterval*' with *Range* or by '*sequenceFixedInterval*' with variable *ListInt*. These alternatives conform with the interval

settings of a test suite described in Section 4.1.3: If the interval is fixed, then the interval until the next test suite is executed after the previous one completed is static and defined in seconds by *Int*. If the time to trigger execution of a test suite is chosen randomly from a range of possible values (also seconds), then this range is defined by *Range*. A special case occurs if each iteration has its own fixed interval, e.g. a test suite which is set to three successive executions where each interval prior to each execution is still fixed, i.e. not chosen randomly, but each interval assumes an individual value. Covering this case in ConTest, the terminal '*sequenceFixedInterval*' with variable *ListInt* is used.

Further, *TestSuite* is defined by the terminals '*Offset*' and '*Timeout*' each of which is followed by the variable *Int*. As described in Section 4.1.3, offset is a fixed time added to the interval between successive test suite runs to avoid successive tests affecting each other. The timeout is the time a test suite run has to successfully complete, otherwise it is interrupted. This is particularly important if external tools such as *Nmap* are used by the continuous test which may have errors that lead to test cases – and thus test suites – not completing.

Finally, *Testsuite* is defined by one or more *TestCase*. The definition of this variable is provided in the following paragraph.

- Lines 39 - 46: *TestCase* is defined by a unique '*TestCaseID*' followed by *ID* as well as '*TestCaseName*' and '*TestCaseModule*' each of which is followed by the variable String. Also, *TestCase* is defined by the optional '*InputParameters*' followed by the variable Parameter. This means that specifying input parameter for a test case may not be required by any implementation of a test case – which is assigned to the variable *String* which follows the terminal '*TestCaseModule*'.
  Finally, *TestCase* is defined by the terminal '*AssertParameters*' which is followed by at least one *Parameter* or more. Having at least one *AssertParameter* is required since the *AssertParameter* is needed to be able to decide whether a test case passed or failed. *Parameter* is defined one or more *KeyValue* whose key is the variable *String* and whose value is either defined by the variable *Int*, *String*, *ListString* or *ListInt*. This corresponds to our definition of test cases provided in Section 4.1.2 where the concept of oracles were introduced, that is, methods determining whether a test case failed or passed. Thus *AssertParameter* specify the input values which are provided to *test oracles*.
- Lines 48 - 71: The variables *Digit*, *Letter*, and *Symbol* are only defined by terminal symbols (Lines 64 - 71). They are used by to construct *Parameter, KeyValue, ListString,*

ListInt, *String*, *ID*, *Int* and *Range* (Lines 48 - 62) which are primitive and composite data types of ConTest.

```
1  ConTest    ::=   Test
2
3  Test       ::=   'TestID' ID
4                   'TestName' String
5                   'TestDescription' String
6                   'Testmetrics'
7                   '{' TestMetric+ '}'
8                   'Workflow'
9                   '{' Workflow '}'
10
11 TestMetric ::=   'TestMetricID' ID
12                  '{'
13                  'TestMetricName' String
14                  'TestMetricModule' String
15                  'Description' String
16                  '}'
17
18 Workflow   ::=   'WorkflowID' ID
19                  'WorkflowName' String
20                  'WorkflowModule' String
21                  'BoundTestsuites'
22                  '{' TestSuite+ '}'
23
24 TestSuite  ::=   'TestSuiteID' ID
25                  '{'
26                  'TestSuiteName' String
27                  'NumberOfMaxIteration' Int | 'infinite'
28                  'IntervalBetweenTests'
29                  '{'
30                  ('fixedInterval' Int
31                  | 'randomizedInterval' Range
32                  | 'sequenceFixedInterval' ListInt)
33                  '}'
34                  'Offset' Int
35                  'Timeout' Int
36                  '{' 'BoundTestCases' TestCase+ '}'
37                  '}'
38
39 TestCase  ::=   'TestCaseID' ID
40                  '{'
41                  'TestCaseName' String
42                  'TestCaseModule' String
43                  'Order' Int
44                  ('InputParameters' '{' Parameter+ '}')?
45                  'AssertParameters' '{' Parameter+ '}'
46                  '}'
47
48 Parameter ::= KeyValue (',' KeyValue)*
49
50 KeyValue  ::= '<' String ':' ( Int | String | ListString | ListInt) '>'
51
52 ListString::= '[' String (',' String)* ']'
53
54 ListInt   ::= '[' Int (',' Int)* ']'
55
56 Range     ::= '[' Int ',' Int ']'
57
58 String    ::= '"'(Letter|Int)(Letter|Int|Symbol)*'"'
59
60 ID        ::= (Letter| Digit) (Letter|'-'| Digit)+
61
62 Int       ::= Digit | Int Digit
63
64 Digit     ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
65
66 Symbol    ::= '('|')'|'<'|'>'|'''|'''|'='|'|'|'.'|','|';'|'\_'
67
68 Letter    ::= 'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'
69                |'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'|'a'|'b'
70                |'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|'o'|'p'
71                |'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'
```

Figure 8: Context-free grammar of ConTest using Extended Backus-Naur Form (EBNF)

Figure 9 shows an exemplary configuration of PortConTest using the language ConTest, one of the exemplary continuous tests described in Section 4.3.

```
1
2   TestID d9ecdea3-e9da-4f78-b041-69b7509e3462
3   TestName PortConTest
4   TestDescription "Continuously tests whether the interfaces of a cloud service component are securely configured."
5   Testmetrics{
6     TestMetricID c5fddea3 {
7       TestMetricName "InsecureConfigurationCounterMetric"
8       TestMetricModule "basicResultCounter"
9       Description "Counts the occurences of failed tests for secure interface configurations."
10    }
11    TestMetricID a5fddea3 {
12      TestMetricName "YearlyInsecureConfigurationMetric"
13      TestMetricModule "cummulativeFailedPassedSequenceDuration"
14      Description "Calculates the ratio between measured insecure interface configuration duration since
15                   test start and 31557600 seconds in a year of 365.25 days."
16    }
17    TestMetricID b5fddea3 {
18      TestMetricName "DailyInsecureConfigurationMetric"
19      TestMetricModule "cummulativeFailedPassedSequenceDuration"
20      Description "Calculates the daily insecure interface configuration duration starting from 00:00 am to 23:59."
21    }
22  }
23
24  Workflow {
25      WorkflowID "PortTestWorkflow"
26      WorkflowName "PortTestwithICMPPreconditionWorkflow"
27      WorkflowModule "BasicIterator"
28      BoundTestsuites {
29        TestSuiteID "PortTestNampTestSuite" {
30          TestSuiteName "SecureInterfaceConfigurationTestSuite"
31            NumberOfMaxIteration infinite
32            IntervalBetweenTests  {
33                randomizeIntervalDuration [30,180]
34                }
35            Offset 15
36            Timeout 300
37            BoundTestCases {
38              TestCaseID PingTest {
39                  TestCaseName "PreConditionPingTestCase"
40                    TestCaseModule "Ping"
41                    Order 1
42                    InputParameters  {
43                     <"count": 10>,
44                     <"host": "10.244.250.9">
45                    }
46                    AssertParameters {
47                     <"round-trip-avg-$lte":"50 ms">,
48                     <"round-trip-sd-$lte":"25 ms">
49                    }
50              }
51
52              TestCaseID PortTest {
53                  TestCaseName "PortTestCase"
54                    TestCaseModule "nmap"
55                    Order 1
56                    InputParameters {
57                     <"host" :"10.244.250.9">
58                    }
59                    AssertParameters {
60                    <"WhiteListedPorts$eq": [80,443,486,993]>,
61                    <"BlackListedPorts$eq": [21,22,25]>
62                    }
63              }
64            }
65        }
66      }
67  }
```

Figure 9: Exemplary continuous test configuration of PortConTest using ConTest

6 IMPLEMENTATION

In order to implement ConTest, the language development tool *XText* is used. This tool is an open source framework to support the development and implementation of domain-specific languages. XText provides various features such as parser generation, code generator or interpreter. Having provided a sound grammar, it generates Eclipse Plugins, thus integrating with the Eclipse IDE and providing editor features such as syntax coloring, code completion and source code navigation.

The next section describes how to define a context-free grammar in XText which uses a notation very similar to EBNF. Thereafter, Section 6.2 present the definition of the XText grammar to generate ConTest. Finally, Section 6.3 describes a code generator which translates the language constructs of ConTest into tool-specific language constructs used by Clouditor.

6.1 GRAMMAR SPECIFICATION WITH XTEXT

XText uses a proprietary language to specify the grammar of a DSL. However, the notation of this language is very similar to EBNF. Hereafter, the characteristics of the language which XText uses to specify a grammar are outlined:

- Each rule consists of a name, a colon, the syntactic form accepted by that rule, and is terminated by a semicolon.
- The semantics of the operators are identical to those the EBNF notation (see paragraph on EBNF in Section 2.2.2).
- The first rule is similar to the start symbol of a grammar in EBNF and defines where the parser starts.
- Keywords of a DSL are defined using terminal string literals which are enclosed with single or double quotes.

XText uses a class model to describe the structure of abstract syntax trees (AST). More specifically, using the Eclipse Modeling Framework (EMF)[17], XText stores a parsed programs as in-memory object graphs. These graphs are instances of EMF Ecore models and represents the

[17] https://eclipse.org/modeling/emf/

AST. Through using these structured data models, XText allows to associate semantics of a meta-model which is supported through the following additional notation:

- XText uses assignment operators to assign consumed information to a feature of the currently produced object. Consider the following example:

  TestSuite:
  'TestSuiteID' tsrID = ID
  ;

  The syntactic declaration for test suites starts with a keyword *'TestSuiteID'* followed by the assignment *tsrId = ID*. The left-hand side points to a feature *tsrId* of the current object. The right hand side *ID* in this case is a rule. It can also be a keyword, a cross-reference (will be explained in the following paragraph) or an alternative which consists of any of these options. An assignment is only valid if the return type of the expression on the right is compatible with the type of the feature. In our above example, ID returns an *EString*, therefore the feature *tsrId* needs to be also of type *EString*.

  Further, there are different types of assignment operators: The assignment operator '=' means that the feature takes exactly one object, '+=' indicates that a feature can be assigned a collection of objects, and '?=' expects a boolean feature, that is, the feature is true if the right-hand side of the assignment was consumed.

- Rules that are enclosed with square brackets "[]" indicate a cross-reference. Cross-referencing means that instead of assigning an object or a collection of objects to a feature, only a reference to one ore more objects of the same type written with the square brackets in the grammar is assigned to a feature. Consider the following example:

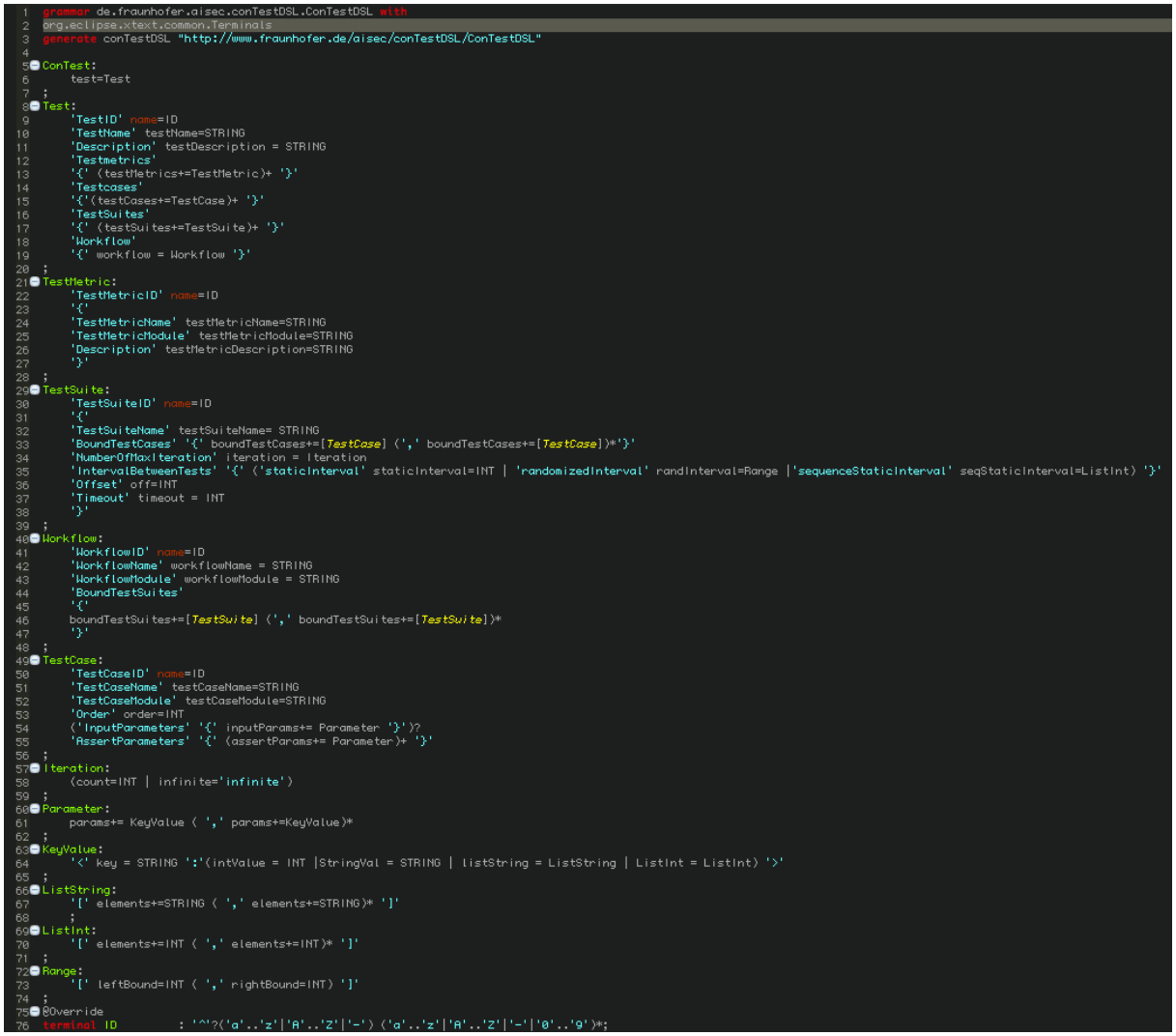
  TestCase :
  'TestCaseID' name = ID
  'TestCaseName' desc = STRING
  ;
  TestSuite :
  'BoundTestCases' boundTestCases += [TestCase]+
  ;

The feature boundTestCases is assigned one or more TestCase objects using a cross-reference. Note that, as a default, XText expects the referred object have to have a feature called *name* which is used for reference.

6.2 IMPLEMENTATION OF CONTEST WITH XTEXT

Figure 10 shows the XText grammar for ConTest. Note that definitions of STRING, and INT are not included in this grammar since these rules are provided by the grammar *org.eclipse.xtext.common.Terminals*, a standard set of terminal rules supplied by XText.

When comparing this grammar with the EBNF representation of ConTest (see Figure 8), it becomes apparent that they are slightly different: In case of the XText representation, the variable *Test* is also defined by at least one *TestSuite* and by at least one TestCase (Lines 15 to 19 of Figure 10) whereas in the EBNF representation, *TestSuite* is part of the definition of the variable Workflow (Line 22 of see Figure 8) and *TestCase* is part of the definition of *TestSuite* (Line 36 of see Figure 8).



```
grammar de.fraunhofer.aisec.conTestDSL.ConTestDSL with
org.eclipse.xtext.common.Terminals
generate conTestDSL "http://www.fraunhofer.de/aisec/conTestDSL/ConTestDSL"

ConTest:
    test=Test
;
Test:
    'TestID' name=ID
    'TestName' testName=STRING
    'Description' testDescription = STRING
    'Testmetrics'
    '{' (testMetrics+=TestMetric)+ '}'
    'Testcases'
    '{'(testCases+=TestCase)+ '}'
    'TestSuites'
    '{' (testSuites+=TestSuite)+ '}'
    'Workflow'
    '{' workflow = Workflow '}'
;
TestMetric:
    'TestMetricID' name=ID
    '{'
    'TestMetricName' testMetricName=STRING
    'TestMetricModule' testMetricModule=STRING
    'Description' testMetricDescription=STRING
    '}'
;
TestSuite:
    'TestSuiteID' name=ID
    '{'
    'TestSuiteName' testSuiteName= STRING
    'BoundTestCases' '{' boundTestCases+=[TestCase] (',' boundTestCases+=[TestCase])*'}'
    'NumberOfMaxIteration' iteration = Iteration
    'IntervalBetweenTests' '{' ('staticInterval' staticInterval=INT | 'randomizedInterval' randInterval=Range |'sequenceStaticInterval' seqStaticInterval=ListInt) '}'
    'Offset' off=INT
    'Timeout' timeout = INT
    '}'
;
Workflow:
    'WorkflowID' name=ID
    'WorkflowName' workflowName = STRING
    'WorkflowModule' workflowModule = STRING
    'BoundTestSuites'
    '{'
    boundTestSuites+=[TestSuite] (',' boundTestSuites+=[TestSuite])*
    '}'
;
TestCase:
    'TestCaseID' name=ID
    'TestCaseName' testCaseName=STRING
    'TestCaseModule' testCaseModule=STRING
    'Order' order=INT
    ('InputParameters' '{' inputParams+= Parameter '}')?
    'AssertParameters' '{' (assertParams+= Parameter)+ '}'
;
Iteration:
    (count=INT | infinite='infinite')
;
Parameter:
    params+= KeyValue ( ',' params+=KeyValue)*
;
KeyValue:
    '<' key = STRING ':'(intValue = INT |StringVal = STRING | listString = ListString | ListInt = ListInt) '>'
;
ListString:
    '[' elements+=STRING ( ',' elements+=STRING)* ']'
    ;
ListInt:
    '[' elements+=INT ( ',' elements+=INT)* ']'
;
Range:
    '[' leftBound=INT ( ',' rightBound=INT) ']'
;
@Override
terminal ID         : '^'?('a'..'z'|'A'..'Z'|'-') ('a'..'z'|'A'..'Z'|'-'|'0'..'9')*;
```

Figure 10: XText grammar definition to generate ConTest

Adapting the EBNF representation of ConTest in the shown manner is only feasible because XText supports cross-referencing of objects. The advantage of this design is that if a developer defines a configuration of a test-based measurement technique, then she has to first specify any test metrics and test cases. Only thereafter can she define the test suites and assign already defined test cases to them. Now, since the developer cannot define a test suite without having defined a test case and bound it to the test suite, the grammar enforces that at least one test suite with one bound test case can be bound to the workflow.

Figure 11 shows an exemplary configuration of the test-based measurement technique *PortConTest* (see Section 4.3.3) using the XText grammar definition of ConTest. Note that this example contains the identical information than Figure 9 which shows an exemplary continuous test definition using the EBNF grammar definition of ConTest. When comparing

Figure 9 and Figure 11, it can be observed that the representation differs. This main reason for this is that the XText grammar supports cross-references which allows to define, e.g., test cases as separate blocks and later reference them using an ID.

6.3 CODE GENERATOR FOR CLOUDITOR

The implementation of ConTest using XText only becomes meaningful if it can be used to generate configurations for specific implementations of test-based measurement techniques. The Clouditor-engine outlined in Section 4.2 which can be used to implement the exemplary continuous test scenarios described in Section 4.3 uses YAML configuration files. Thus a code generator has to be implemented which translates the constructs of ConTest to sound YAML constructs which can be consumed by the Clouditor-engine.

In order to generate application code, XText uses *XTend* which is a dialect of Java. Xtend provides multi-line template expressions which a developer can use to write strings representing parts of the code to be generated. Syntactically, these template expressions are defined by enclosing triple single quotes.

Upon generating language artifacts with XText for a particular grammar, the code generator stub is automatically supplied. As mentioned above, XText uses EMF Ecore models to store parsed programs as object graphs. These models serve as input to the code generator where the object graph is contained in an Ecore Resource Object. In order to generate the desired code, the compile method of the code generator has to be implemented.

Figure 12 shows an extract of the code generator's compile method that was implemented to generate YAML files which are consumed by the Clouditor-Engine in order to configure the

exemplary continuous tests described in Section 4.3.3. Figure 13 shows the YAML representation of *PortConTest* defined in ConTest shown in Figure 11.

```
TestID d9ecdea3-e9da-4f78-b041-69b7509e3462
TestName "PortConTest"
Description "Continuously tests whether the interfaces of a cloud service component are securely configured"

Testmetrics {
        TestMetricID c5fddea3 {
        TestMetricName "InsecureConfigurationCounterMetric"
        TestMetricModule "basicResultCounter"
        Description "Counts the occurences of failed tests for secure interface configurations."
        }

        TestMetricID a5fddea3 {
        TestMetricName "YearlyInsecureConfigurationMetric"
        TestMetricModule "cummulativeFailedPassedSequenceDuration"
        Description "Calculates the ratio between measured insecure interface configuration duration since test start and 31557600 seconds in a year
                    of 365.25 days."
        }

        TestMetricID b5fddea3 {
            TestMetricName "DailyInsecureConfigurationMetric"
            TestMetricModule "cummulativeFailedPassedSequenceDuration"
            Description "Calculates the daily insecure interface configuration duration starting from 00:00 am to 23:59."
        }
}

Testcases {
    TestCaseID PingTest
    TestCaseName "PreConditionPingTestCase"
    TestCaseModule "Ping"
    Order 1
    InputParameters {
        <"count":10>,
        <"host":"10.244.250.9">
    }
    AssertParameters {
        <"round-trip-avg-$lte":"75 ms">,
        <"round-trip-sd-$lte":"25 ms">
    }

    TestCaseID PortTest
    TestCaseName "PortTestCase"
    TestCaseModule "nmap"
    Order 1
    InputParameters {
        <"host":"10.244.250.9">
    }
    AssertParameters {
        <"WhiteListedPorts$eq": [80,443,486,993]>,
        <"BlackListedPorts$eq": [21,22,25]>
    }
}

TestSuites
{
    TestSuiteID PortTestNampTestSuite
    {
        TestSuiteName "SecureInterfaceConfigurationTestSuite"
        BoundTestCases {PingTest, PortTest}
        NumberOfMaxIteration infinite
        IntervalBetweenTests {
            randomizedInterval [30,180]
        }
        Offset 15
        Timeout 300
        }
    }
    Workflow {
        WorkflowID PortTestWorkflow
        WorkflowName "PortTestwithICMPPreconditionWorkflow"
        WorkflowModule "BasicIterator"
        BoundTestSuites {PortTestNampTestSuite}
}
```

Figure 11: Exemplary continuous test configuration of PortConTest using ConTest build with XText grammar

```
def compile(Test ct) '''
    «var length2 = "  "»
    «var length4 = length2 + "  "»

    name: «ct.testName»
    id: «ct.name»
    description: «ct.testDescription»

    metrics:
    «FOR m : ct.testMetrics»
    «length2»- class: « m.testMetricModule»
    «length4»name: « m.testMetricName»
    «length4»description: « m.testMetricDescription»
    «ENDFOR»

    testCases:
    «FOR tc : ct.testCases»
    «length2»«tc.testCaseName»:
    «length4»'@id': «tc.name»
    «length4»order: «tc.order»
    «IF tc.inputParams !=null»
        «FOR ip : tc.inputParams»
            «FOR kv : ip.params»
                «length4»«kv.key»: «identifyParams(kv)»
            «ENDFOR»
        «ENDFOR»
    «ENDIF»
    «FOR ap : tc.assertParams»
        «FOR kv : ap.params»
            «length4»«kv.key»: «identifyParams(kv)»
        «ENDFOR»
    «ENDFOR»

    «ENDFOR»
    workflow:
    «length2»class: «ct.workflow.workflowModule»
    «length2»name: «ct.workflow.workflowName»
    «length2»testSuites:
    «FOR ts : ct.testSuites»
    «length4»«ts.name»:
    «length4»name: «ts.name»
    «length4»label: «ts.testSuiteName»
    «length4»randomized: «IF ts.randInterval != null»true «ELSE»false «ENDIF»
    «length4»iteration: «IF ts.iteration.infinite[==null»-1«ELSE»«ts.iteration.count»«ENDIF»
    «length4»interval: «IF ts.randInterval != null»[«ts.randInterval.leftBound»,«ts.randInterval.rightBound»]«ELSE»«IF ts.seqStaticInterval !=null»«ts.seqStaticInterval.elements»«ELSE»[«ts.staticInterval»]«ENDIF»«ENDIF
    «length4»offset: «ts.off»
    «length4»timeout: «ts.timeout»
    «length4»testCases: [«FOR tc:ct.testCases SEPARATOR ','»'@ref': «tc.name»«ENDFOR»]
    «ENDFOR»
'''
```

Figure 12: Compile method of XText code generator to translate ConTest to YAML (used to configure Clouditor-Engine)

```yaml
name: PortConTest
id: d9ecdea3-e9da-4f78-b041-69b7509e3462
description: Continuously tests whether the interfaces of a cloud service component are securely configured

metrics:
  - class: basicResultCounter
    name: InsecureConfigurationCounterMetric
    description: Counts the occurences of failed tests for secure interface configurations.
  - class: cummulativeFailedPassedSequenceDuration
    name: YearlyInsecureConfigurationMetric
    description: Calculates the ratio between measured insecure interface configuration duration since test start and 31557600 seconds in a year
        of 365.25 days.
  - class: cummulativeFailedPassedSequenceDuration
    name: DailyInsecureConfigurationMetric
    description: Calculates the daily insecure interface configuration duration starting from 00:00 am to 23:59.

testCases:
  PreConditionPingTestCase:
    '@id': PingTest
    order: 1
    count: 10
    host: 10.244.250.9
    round-trip-avg-$lte: 75 ms
    round-trip-sd-$lte: 25 ms

  PortTestCase:
    '@id': PortTest
    order: 1
    host: 10.244.250.9
    WhiteListedPorts$eq: [80, 443, 486, 993]
    BlackListedPorts$eq: [21, 22, 25]

workflow:
  class: BasicIterator
  name: PortTestwithICMPPreconditionWorkflow
  testSuites:
    PortTestNampTestSuite:
    name: PortTestNampTestSuite
    label: SecureInterfaceConfigurationTestSuite
    randomized: true
    iteration: -1
    interval: [30,180]
    offset: 15
    timeout: 300
    testCases: ['@ref': PingTest,'@ref': PortTest]
```

Figure 13: YAML file generated from ConTest to configure PortConTest with Clouditor-Engine

7 CONCLUSION

In this deliverable, a universal configuration language called ConTest was introduced which permits to represent the production of measurement result in a general manner. This implies that ConTest is agnostic to specific implementations of test-based measurement techniques. Furthermore, ConTest serves as starting point from which specific configurations of a measurement technique can be automatically generated. This ensures that the configuration of a measurement technique deployed to produce some measurement results adheres to the domain concepts of continuous test-based measurement, a crucial part of continuous security audits.

In order to develop ConTest, the process provided by Mernik et al. (17) was followed which describes the necessary steps to develop a domain specific language (DSL). These include:

- *Motivate the necessity to build a DSL*: The purpose of ConTest is to provide a general representation of a test-based measurement technique's configuration which is agnostic to a concrete implementation but adheres to the building blocks of continuous test-based measurements introduced in Section XXX. This not only allows to rigorously compare the measurement results produced by different implementations of test-based techniques but also ensures conformance with the building blocks by a developer having to provide a code generator with which configurations written in ConTest can be translated into the target configuration language used by a specific implementation of a test-based measurement technique.

- *Analysis of domain-specific constructs*: In this step, it was investigated which parts of the building blocks are suitable to be used for a general representation of test-based measurement techniques' configuration, including test cases, test suites, workflow, test metrics and preconditions.

- *Design ConTest*: ConTest was defined using Extended Backus-Naur-Form (EBNF), a (domain specific) language to describe context-free grammars.

- *Implement ConTest*: In order to implement ConTest, the language development tool *XText* was used which uses a variant of EBNF to define context-free grammars. Furthermore, a code generator was implemented which translates the constructs of

ConTest to YAML configuration files which are used within the implementation of all exemplary continuous test scenarios by Clouditor presented in Chapter 4.3.

8 REFERENCES

1. **The German Federal Office for Information Security (BSI).** *Cloud Computing Compliance Controls Catalogue (C5).* https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/CloudComputing/ComplianceControlsCatalogue/ComplianceControlsCatalogue.pdf?__blob=publicationFile&v=4 : s.n.
2. **Cloud Security Alliance (CSA).** Security, Trust and Assurance Registry (STAR). [Online]
3. **Khan, Khaled M and Malluhi, Qutaibah.** Establishing trust in cloud computing. *IT professional.* 2010, Bd. 12, 5, S. 20-27.
4. **Ko, Ryan KL and Jagadpramana, Peter and Mowbray, Miranda and Pearson, Siani and Kirchberg, Markus and Liang, Qianhui and Lee, Bu Sung.** TrustCloud: A framework for accountability and trust in cloud computing. *7th IEEE World Congress on Services (SERVICES).* 2011, S. 584-588.
5. **Sunyaev, Ali and Schneider, Stephan.** Cloud services certification. *Communications of the ACM.* 2013.
6. **Cimato, Stelvio and Damiani, Ernesto and Zavatarelli, Francesco and Menicocci, Renato.** Towards the certification of cloud services. *9th IEEE World Congress on Services (SERVICES).* 2013.
7. **Windhorst, Iryna and Sunyaev, Ali.** Dynamic certification of cloud services. *8th International Conference on Availability, Reliability and Security (ARES).* 2013.
8. **Stephanow, Philipp and Banse, Christian and Schütte, Julian.** Generating Threat Profiles for Cloud Service Certification Systems. *17th IEEE High Assurance Systems Engineering Symposium (HASE).* 2016.
9. **Kaliski Jr, Burton S and Pauley, Wayne.** Toward risk assessment as a service in cloud environments. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing.* 2010.
10. **Krotsiani, Maria and Spanoudakis, George and Mahbub, Khaled.** Incremental certification of cloud services. *7th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE).* 2013.
11. **Lins, Sebastian and Schneider, Stephan and Sunyaev, Ali.** Trust is Good, Control is Better: Creating Secure Clouds by Continuous Auditing. *Transactions on Cloud Computing.* 2016.
12. **Hopcroft, John E and Motwani, Rajeev and Ullman, Jeffrey D.** Automata theory, languages, and computation. *International Edition.* 2006.
13. **Stephanow, Philipp and Fallenbeck, Niels.** Towards continuous certification of Infrastructure-as-a-Service using low-level metrics. *12th IEEE International Conference on Advanced and Trusted Computing (ATC).* 2015.
14. **Schiffman, Joshua and Sun, Yuqiong and Vijayakumar, Hayawardh and Jaeger, Trent.** Cloud verifier: Verifiable auditing service for iaas clouds. *9th IEEE World Congress on Services (SERVICES).* 2013, S. 239-246.
15. **Stephanow, Philipp and Gall, Mark.** Language Classes for Cloud Service Certification Systems. *11th IEEE World Congress on Services (SERVICES).* 2015.
16. **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, S. 1117-1126.

17. **Stephanow-Gierach, Philipp.** Continuous test-based certification of cloud services. PhD Thesis. Under review, 2017.
18. **Mernik, Marjan and Heering, Jan and Sloane, Anthony M.** When and how to develop domain-specific languages. *ACM computing surveys (CSUR).* 2005, Bd. 37, 4, S. 316-344.
19. **Fowler, Martin.** Domain-specific languages.
20. **Chomsky, Noam.** On certain formal properties of grammars. *Information and control.* 1959, Bd. 2, 2, S. 137-167.
21. **Backus, John W.** The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. *Proceedings of the International Comference on Information Processing.* 1959.
22. **Pattis, Richard E.** Ebnf: A notation to describe syntax. 2013.
23. **Wirth, Niklaus.** Extended backus-naur form (EBNF). *ISO/IEC.* 1996.
24. **W3C.** Extensible Markup Language (XML) 1.0 (Third Edition): Section Notation: The formal grammar of XML using Extended Backus-Naur Form (EBNF) notation. 2004.
25. **Internet Engineering Task Force (IETF).** RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2. 2008.
26. **IETF.** RFC 7465: Prohibiting RC4 Cipher Suites. 2015.
27. **Cloud Security Alliance (CSA).** Cloud Control Matrix: Security Controls Framework for Cloud Providers & Consumers. 2015.
28. **International Organization for Standardization (ISO).** ISO/IEC 27001:2013 Information technology -- Security techniques -- Information security management systems -- Requirements.
29. **Mell, Peter and Grance, Timothy.** The NIST Definition of Cloud Computing. *NIST Special Publication.* 2011.
30. **Grobauer, Bernd and Walloschek, Tobias and Stöcker, Elmar.** Understanding cloud computing vulnerabilities. *Security & privacy, IEEE.* 2011, Bd. 9, 2, S. 50-57.
31. **The OWASP Foundation.** OWASP Top 10 - 2013: The Ten Most Critical Web Application Security Risks. *The Open Web Application Security.* 2013.
32. **Halfond, William G and Viegas, Jeremy and Orso, Alessandro.** A classification of SQL-injection attacks and countermeasures. *Proceedings of the International Symposium on Secure Software Engineering.* 2006, Bd. 1, S. 13-15.