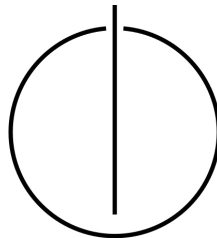
Technische Universität
München

Continuous Test-Based Certification of Cloud Services

Philipp Stephanow-Gierach



Dissertation

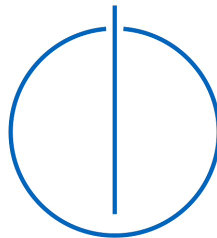
Technische Universität München

Lehrstuhl für Sicherheit in der Informatik

Fakultät für Informatik

Continuous Test-Based Certification of Cloud Services

Philipp Stephanow-Gierach



Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Jens Großklags	
Prüfer der Dissertation:	1.	Prof. Dr. Claudia Eckert
	2.	Prof. Dr. Ali Sunyaev, Karlsruher Institut für Technologie (KIT)

Die Dissertation wurde am 13.06.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.11.2018 angenommen.

Acknowledgments

Over the last years, a lot of people have supported and encouraged me to work on this thesis. Without them, it simply would not have been possible. To begin with, I want to thank my supervisor Prof. Dr. Claudia Eckert, head of Fraunhofer AISEC and the Chair for IT Security at the Technical University of Munich (TUM). Her confidence in my scientific abilities served as the crucial ingredient to prepare this thesis. She was always available for constructive, valuable feedback and gave me the opportunity to further advance my ideas. Furthermore, I would like to thank Prof. Dr. Ali Sunyaev, head of the Critical Information Infrastructures research group and professor for Applied Computer Science at the Karlsruhe Institute of Technology (KIT), for kindly offering a second opinion on this thesis.

Also, I would like to express my gratitude to Dr. Julian Schütte, not only for many hours of fruitful discussions but also for providing guidance how to balance between work and private life. Moreover, I want to thank all of my former and present colleagues from Fraunhofer AISEC with whom I worked on topics related to this thesis. In alphabetical order, I want to thank: Christian Banse, Dr. Niels Fallenbeck, Mark Gall, Mario Hoffmann, Immanuel Kunz, Mohammad Moein, Gaurav Srivastava and Iryna Windhorst.

Further, I have to thank my mother who, after all, paved the way for me to pursue this academic path. Her ambition, discipline and tenacity always have and always will inspire me.

Finally, I want to thank my wife Jule who has been the most patient companion: Time and space you granted me to do this work extend well beyond generally acceptable limits. Many people supported me to advance this thesis, however, without *you*, I could not have completed it.

Kurzfassung

Cloud-Dienste ermöglichen Nutzern den flexiblen, bedarfsabhängigen Einsatz scheinbar unbegrenzter IT-Ressourcen. Diese Art der Ressourcennutzung verspricht Kostensenkungen, birgt aber für Cloud-Nutzer auch Risiken, etwa im Hinblick auf Sicherheit, Zuverlässigkeit sowie Einhaltung rechtlicher Vorgaben. Diese Risiken hindern Unternehmen sowie staatliche Einrichtungen daran, die wirtschaftlichen Vorteile von Cloud-Diensten auszuschöpfen. Ein Ansatz um den Einsatz von Cloud-Diensten zu fördern, besteht in der *Zertifizierung* dieser Dienste. Dadurch wird gezeigt, dass ein Cloud-Dienst eine Menge spezieller Anforderungen, sogenannte *Controls*, erfüllt. Allerdings eignet sich der traditionelle Ansatz der Zertifizierung nicht für Cloud-Dienste: Traditionellerweise werden die Anforderungen eines Zertifikates zu einem bestimmten Zeitpunkt geprüft, wobei die Ergebnisse einer erfolgreichen Prüfung für einen bestimmten Zeitraum gültig sind, in der Regel für ein bis drei Jahre. Die Eigenschaften eines Cloud-Dienstes können sich allerdings im Gültigkeitszeitraum des Zertifikates derart ändern, so dass der Dienst Anforderungen des Zertifikates nicht mehr erfüllt. Das Zertifikat ist somit faktisch nicht mehr gültig. Folglich bedarf die Zertifizierung von Cloud-Diensten eines neuen Ansatzes, der *kontinuierlich* die Veränderungen eines Dienstes zur Betriebszeit detektiert und die Auswirkungen dieser Veränderungen auf die Erfüllung der Anforderungen eines Zertifikates bewertet. Aktuelle Forschungsansätze schlagen daher vor, die Zertifizierung von Cloud-Diensten zu automatisieren. Diesen Ansätzen ist gemein, dass sie strukturelle Änderungen der Infrastruktur des zu zertifizierenden Cloud-Dienstes voraussetzen. Ferner berücksichtigen diese Ansätze nicht die *wiederholte* und *selbst-adaptierende* Ausführung automatisierter Anforderungsprüfungen und gehen grundsätzlich davon aus, dass der Anbieter eines Cloud-Dienstes nicht versucht, Anforderungsprüfungen zu manipulieren. Vor diesem Hintergrund leistet die vorliegende Arbeit folgende Beiträge: Es wird ein Framework zur Erstellung minimalinvasiver Tests entwickelt, die die kontinuierliche Zertifizierung von Cloud-Diensten unterstützen. Die Anwendung des Frameworks wird anhand von fünf Beispielszenarien demonstriert. Ferner wird die Konfigurationssprache *ConTest* vorgestellt, die eine einheitliche, implementierungsunabhängige Repräsentation von Tests auf Basis der Bausteine des Frameworks ermöglicht. ConTest kann genutzt werden, um Konfigurationen für konkrete Testimplementierungen zu generieren. Weiterhin wird in dieser Arbeit ein Vorgehen vorgestellt, um die Genauigkeit kontinuierlicher Tests experimentell zu untersuchen. Es wird gezeigt, wie mit diesem Ansatz die Genauigkeit von Testalternativen sowie alternativen Testkonfigurationen verglichen werden kann. Schließlich wird in dieser Arbeit ein Modell für einen betrügerischen Anbieter von Cloud-Diensten vorgestellt, der nur vorgibt, Anforderungen eines Zertifikates zu erfüllen, tatsächlich aber betrügt, wenn er davon ausgeht, dass dieser Betrug nicht durch Tests festgestellt wird. Auf Basis dieses Modells wird gezeigt, wie die zufällige Auswahl von Testparametern den Anreiz dieses Anbieters verringert zu betrügen und es wird dargelegt, wie das Framework diese Art der Konfiguration von Tests unterstützt.

Abstract

What makes cloud services attractive to users is having a contractual framework within which they have access to seemingly unlimited IT resources provided as a service instead of being bound to specific resources meticulously negotiated and paid for upfront. Yet, while cloud services promise cost reductions through flexible, on-demand usage of IT resources, using cloud services also entails risks related to, e.g., security, reliability and legal regulations since part of internal business processes are now supported through external IT resources. These risks hinder wide-spread adoption and inhibit companies and public institutions to fully embrace the economic benefits that cloud services offer. One approach to foster adoption of cloud services is *certification*, that is, demonstrating that a cloud service complies with a set of particular requirements called *controls*. However, the traditional concept of certification cannot be applied to cloud services because, traditionally, controls of a certificate are evaluated at some point in time and, if successful, a certificate is issued usually valid in the range from one to three years. A cloud service, however, may change over time and thus not fulfill one or more certificate's controls anymore, thereby rendering the certificate invalid. Therefore, cloud service certification requires a different approach capable of *continuously*, i.e., *automatically* and *repeatedly* detecting ongoing changes of a cloud service during operation and assessing their impact on satisfaction of a certificate's controls. Recent research proposes automated means supporting cloud service certification. These approaches have in common that they are invasive by design, that is, they require structural changes to the infrastructure of the cloud service under certification. Also, they do not incorporate the notion of *repeated* and *self-adaptive* execution of automated checks and generally assume that the cloud service provider does not attempt to manipulate checks. In order to address these gaps, this thesis introduces a framework to guide the design of minimally invasive tests which support continuous cloud service certification. We present five example scenarios where we implement tests following our framework and show how these tests support continuous certification. Moreover, we present a configuration language called *ConTest* which provides a general, unified representation of a continuous test which is agnostic to a concrete implementation. ConTest can be used to generate test definitions to configure any specific test implementation. We also introduce a method to experimentally evaluate the accuracy and precision of continuous test results. We demonstrate the application of this method with three example scenarios where we evaluate accuracy and precision of alternative tests as well as alternative test configurations to select the most suitable one. Finally, the last contribution of this thesis is a model of an adversarial cloud service provider who only pretends to comply with a set of controls but cheats if he is sure that he is not caught through testing. Based on this model, we show that randomization of tests reduces the willingness of the adversarial provider to cheat and point out how our framework to design tests supports randomization on different levels.

Contents

List of Figures xiii

List of Tables xv

1 Introduction 1
- 1.1 Motivation and problem statement 1
- 1.2 Research challenges 4
  - 1.2.1 Challenge 1: Design of tests supporting continuous cloud certification 4
  - 1.2.2 Challenge 2: Definition of continuous tests 5
  - 1.2.3 Challenge 3: Accuracy and precision of continuous test results 6
  - 1.2.4 Challenge 4: Trustworthy continuous test results 7
- 1.3 Contributions 8
- 1.4 Organization of this thesis 9

2 Background 13
- 2.1 Cloud computing and cloud services 13
  - 2.1.1 Cloud service models 14
  - 2.1.2 Deployment models 15
  - 2.1.3 Cloud services in context of this thesis 16
- 2.2 Testing 16
  - 2.2.1 Software testing 16
  - 2.2.2 Security testing 20
  - 2.2.3 Conformance testing 21
  - 2.2.4 Testing in context of this thesis 22
- 2.3 Certification 23
  - 2.3.1 Software certification 24
  - 2.3.2 Service certification 25
  - 2.3.3 Cloud service certification 26
  - 2.3.4 Certification in context of this thesis 30

3 Related Work 31
- 3.1 Test-based cloud service certification models 31
  - 3.1.1 Frameworks to support test-based certification 32
  - 3.1.2 Specialized test-based evidence production methods 33
  - 3.1.3 Comparative benchmark testing of cloud services 35
- 3.2 Other cloud service certification models 36
  - 3.2.1 Monitoring-based certification models 36

3.2.2 TPM-based certification models 39
3.3 Testing as a service 39
3.4 Summary and identification of gaps 41

4 A framework to support continuous test-based cloud service certification 45
4.1 Approach 45
4.2 Requirements 46
4.2.1 Production of evidence 46
4.2.2 Extensibility 46
4.2.3 Flexible integration with existing infrastructures of cloud services . 47
4.2.4 Independence of cloud service model 48
4.2.5 Reusability of continuous test components 48
4.2.6 Integration of existing test tools 48
4.2.7 Self-adaptivity 49
4.3 Framework building blocks 49
4.3.1 Overview 49
4.3.2 Test cases 50
4.3.3 Test suites 52
4.3.4 Workflow 53
4.3.5 Test metrics 54
4.3.6 Preconditions 57
4.4 Example implementation of the framework 61
4.4.1 Clouditor's engine 61
4.4.2 Clouditor's dashboard 62
4.5 Summary and discussion 65

5 Example continuous test scenarios 69
5.1 General characteristics of continuous test scenarios 70
5.2 Continuously testing availability 72
5.2.1 Property description and overview of test scenario 72
5.2.2 General characteristics of the test scenario 73
5.2.3 Test design 75
5.2.4 Implementation and experiment 79
5.3 Continuously testing location 80
5.3.1 Property description and overview of test scenario 80
5.3.2 General characteristics of the test scenario 83
5.3.3 Test design 85
5.3.4 Implementation and experiment 92
5.4 Continuously testing secure communication configuration 98
5.4.1 Property description and overview of test scenario 98
5.4.2 General characteristics of the test scenario 99
5.4.3 Test design 100
5.4.4 Implementation and experiment 102
5.5 Continuously testing secure interface configuration 104
5.5.1 Property description and overview of test scenario 105
5.5.2 General characteristics of the test scenario 105
5.5.3 Test design 107

5.5.4 Implementation and experiment 108
5.6 Continuously testing user input validation 110
5.6.1 Property description and overview of test scenario 110
5.6.2 General characteristics of the test scenario 111
5.6.3 Test design 113
5.6.4 Implementation and experiment 114
5.7 Summary and discussion 116

6 Definition of continuous tests 119
6.1 Developing domain-specific languages 120
6.1.1 DSL engineering 120
6.1.2 Formal languages 121
6.2 Continuous test definition language 123
6.2.1 Decision 123
6.2.2 Analysis 123
6.2.3 Design 125
6.2.4 Implementation 129
6.3 Summary and discussion 136

7 Evaluation of continuous test results 137
7.1 Accuracy and precision of continuous test results 138
7.1.1 Accuracy 138
7.1.2 Precision 139
7.2 Overview of the evaluation process 142
7.3 Control violations 143
7.3.1 Control violation sequence 143
7.3.2 Control violation design 144
7.3.3 Standardizing control violation events 145
7.4 Accuracy and precision measures 145
7.4.1 Basic-Result-Counter 146
7.4.2 Failed-Passed-Sequence-Counter 153
7.4.3 Failed-Passed-Sequence-Duration 157
7.4.4 Cumulative-Failed-Passed-Sequence-Duration 162
7.5 Example evaluation of continuous test results 164
7.5.1 Setup and environment 164
7.5.2 Continuously testing availability 165
7.5.3 Continuously testing secure communication configuration 174
7.5.4 Continuously testing secure interface configuration 180
7.6 Summary and discussion 189

8 Test-based certification of opportunistic cloud service providers 193
8.1 Background 194
8.1.1 Covert adversaries 194
8.1.2 Labeled Transition Systems 196
8.2 Opportunistic provider as a labeled transition system 197
8.2.1 States 198
8.2.2 Atomic propositions and labeling function 200
8.2.3 Actions 200

8.2.4 Choice of behavior 201
8.3 Randomization of tests as a countermeasure 203
8.3.1 Effect of randomization on opportunistic behavior 203
8.3.2 Framework support of randomized tests 205
8.4 Summary and discussion 206

9 Conclusion 209
9.1 Contributions to research challenges 209
9.2 Directions for future research 211

Bibliography 215

Acronyms 239

A External tool configurations 243

B Code snippets 245

C Publications in the context of this thesis 247

List of Figures

2.1 Dimensions of testing types (based on Figures from [1][2]) 18

4.1 Overview of building blocks to support continuous test-based certification of cloud services 50
4.2 Failed-Passed-Sequence (fps) example based on basic test results (br) . . . 55
4.3 Continuously executed tests (tsr) with universal test metric $fpsD$ 56
4.4 Extract of an example continuous test using a specialized test suite to test preconditions before executing the main test suite 58
4.5 Extract of an example continuous test using precondition test cases as part of the main test suites 59
4.6 Bounds of a *Preconditions-Failed-Sequence-Duration* ($pfsD$) 60
4.7 Components of the Clouditor toolbox 61
4.8 Landing page of Clouditor's dashboard 63
4.9 Different visualizations of test suite results and test metrics of example continuous test *Availability VM1* 64

5.1 Workflow of continuously testing resource availability (CT^{AV}) 77
5.2 Extract of a test sequence to estimate downtime of a cloud service component 78
5.3 Feature vector derived from measurements on Internet and Transport layer provided by TC^{IPD} and TC^{TCPD} 86
5.4 Workflow of continuously validating the locations of cloud service components (CT^{LV}) 90
5.5 AWS Global Infrastructure (Figure based on [3]) 93
5.6 Evolving test accuracy per batch over time of CT^{LV} 95
5.7 Filtered outliers per batch of CT^{LV} 96
5.8 Evolving training error ε observed per batch over time of CT^{LV} 97
5.9 Adaption of invalidation window size per batch over time of CT^{LV} 97
5.10 Workflow of continuously testing secure communication configuration (CT^{SC}) 102
5.11 Workflow of continuously testing secure interface configuration (CT^{SI}) . . 108
5.12 Workflow of continuously testing user input validation (CT^{UI}) 114

7.1 Experimental evaluation of the accuracy and precision of continuous test results 143
7.2 Sequence of control violation events cve 144
7.3 True negative basic test result (br^{TN}) 146
7.4 True positive basic test result (br^{TP}) 147
7.5 False negative basic test result (br^{FN}) 148
7.6 False positive basic test result (br^{FP}) 148

7.7 False positive basic test result (br^{FP}) covering multiple cve 148
7.8 Pseudo true negative basic test result (br^{PTN}) where the test ends during cve 149
7.9 Pseudo true negative basic test result (br^{PTN}) where the test starts during cve 150
7.10 True negative Failed-Passed-Sequence (fps^{TN}) 153
7.11 False negative fps 154
7.12 False negative Failed-Passed-Sequence (fps^{FN}) with true negative and false negative basic test result (br^{TN} & br^{FN}) 154
7.13 False negative Failed-Passed-Sequence (fps^{FN}) with false positive basic test result (br^{FP}) 155
7.14 False positive fps 155
7.15 True negative Failed-Passed-Sequence-Duration ($fpsD^{TN}$) which overestimates total duration of cve_i and cve_{i+1} 158
7.16 True negative Failed-Passed-Sequence-Duration ($fpsD^{TN}$) which underestimates duration of cve_i 158
7.17 True negative Failed-Passed-Sequence-Duration ($fpsD^{TN}$) with $efpsD^{TN}_{pre} > 0$ and $efpsD^{TN}_{post} > 0$ 159
7.18 False negative Failed-Passed-Sequence-Duration ($fpsD^{FN}$) 159
7.19 False positive Failed-Passed-Sequence-Duration ($fpsD^{FP}$) 160
7.20 Duration error of true negative fps ($efpsD^{TN}$) of PingTest, TCPTest, and SSHTest 171
7.21 Total duration error of true negative fps ($ecfpsd^{TN}$) and false negative fps ($ecfpsd^{FN}$) of PingTest, TCPTest, and SSHTest 173
7.22 Relative duration error of true negative fps ($efpsD^{TN}_{rel}$) of TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$ and TLSTest$^{[0,60]}$ 182
7.23 Selected accuracy measures for PortTest^{10}, PortTest^{30} and PortTest^{60} based on the Basic-Result-Counter (brC) test metric 186
7.24 Selected accuracy measures for PortTest^{10}, PortTest^{30} and PortTest^{60} based on the Failed-Passed-Sequence-Counter ($fpsC$) test metric 189

8.1 Example LTS OCP^{VM} consisting of three states 202

List of Tables

4.1 Options to define Failed-Passed-Sequence-Duration ($fpsD$) if $|fps| > 2$ . . 56

5.1 Test statistics and results of continuous test CT^{AV} 81
5.2 Results of continuous location validation of 14 AWS EC2 instances using 10% of total data set as initial training data and 898 successive batches with each 140 newly collected probes 97
5.3 Test statistics and results of continuous test CT^{SC} 104
5.4 Test statistics and results of continuous test CT^{SI} 110
5.5 Test statistics and results of continuous test CT^{UI} 116

7.1 Summary of control violation statistics for PingTest, TCPTest, and SSHTest 167
7.2 Summary of test statistics of PingTest, TCPTest, and SSHTest 168
7.3 Evaluation of PingTest, TCPTest and SSHTest to test availability of $IaaS^{OS}$ based on the Basic-Result-Counter (brC) test metric 170
7.4 Evaluation of PingTest, TCPTest and SSHTest to test availability of $IaaS^{OS}$ based on the Failed-Passed-Sequence-Counter ($fpsC$) test metric 171
7.5 Evaluation of PingTest, TCPTest and SSHTest to test availability of $IaaS^{OS}$ based on the Failed-Passed-Sequence-Duration ($fpsD$) test metric 172
7.6 Evaluation of PingTest, TCPTest and SSHTest to test availability of $IaaS^{OS}$ based on the Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) test metric 173
7.7 Summary of control violation sequence statistics for $\text{TLSTest}^{[0,10]}$, $\text{TLSTest}^{[0,30]}$, and $\text{TLSTest}^{[0,60]}$ 176
7.8 Summary of test statistics of $\text{TLSTest}^{[0,10]}$, $\text{TLSTest}^{[0,30]}$, and $\text{TLSTest}^{[0,60]}$. 177
7.9 Evaluation of TLSTest to test secure communication configuration of $SaaS^{OS}$ based on the Basic-Result-Counter (brC) test metric 179
7.10 Evaluation of TLSTest to test secure communication configuration of $SaaS^{OS}$ based on the Failed-Passed-Sequence-Counter ($fpsC$) test metric 180
7.11 Evaluation of TLSTest to test secure communication configuration of $SaaS^{OS}$ based on the Failed-Passed-Sequence-Duration ($fpsD$) test metric 181
7.12 Evaluation of TLSTest to test secure communication configuration of $SaaS^{OS}$ based on the Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) test metric 182
7.13 Summary of control violation sequence statistics for PortTest^{10}, PortTest^{30}, and PortTest^{60} 183
7.14 Summary of test statistics of PortTest^{10}, PortTest^{30}, and PortTest^{60} 184

7.15 Evaluation of PortTest to test secure interface configuration of $PaaS^{OS}$ based on the Basic-Result-Counter (brC) test metric 187
7.16 Evaluation of PortTest to test secure interface configuration of $PaaS^{OS}$ based on the Failed-Passed-Sequence-Counter ($fpsC$) test metric 187
7.17 Evaluation of PortTest to test secure interface configuration of $PaaS^{OS}$ based on the Failed-Passed-Sequence-Duration ($fpsD$) test metric 188
7.18 Evaluation of PortTest to test secure interface configuration of $PaaS^{OS}$ based on the Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) test metric . 189

Chapter 1

Introduction

The *Babbage Principle* introduced by Charles Babbage, the inventor of the first special-purpose computer [4], states that in order to reduce labor costs to produce a certain artifact, *labor* has to be divided according to the skill of the workers: Only difficult tasks should be completed by highly skilled (and high-paid) workers whereas less difficult tasks are assigned to less skilled, less paid workers [5]. The Babbage Principle is one description of the *division of labor*, the fundamental principle underlying industrial organization and production [6].

While the division of labor is an abstract concept of economic theory, we can treat *outsourcing* as one example observable in the real world: Here, companies or government agents assign tasks which are part of their business processes to other, external companies in order to reduce costs [7]. The rationale behind outsourcing employs principles of the division of labor through trading specialized capabilities. *IT outsourcing* is a special type of outsourcing where companies – driven by the intention to refocus on core competencies and perceiving internal IT departments and services as mere cost centers – contract out IT services to external vendors [8]. *Cloud service providers* are a special type of such IT service vendors [9] which offer *cloud services*, that is, IT resources such as virtual machines, platform services as well as ready-to-use applications as scalable services which are accessible over the Internet and billed on a pay-per-use basis [10].

What makes cloud services attractive to customers and superior to traditional IT outsourcing is having a contractual framework within which the customer has access to seemingly unlimited IT resources provided as a service instead of being bound to specific resources meticulously negotiated and paid for upfront [11]. Furthermore, the economic incentive from a cloud provider's point of view originates from the economy of scale: Through constructing and operating large-scale data centers at low-cost locations, cloud providers can supply IT resources and services below the costs of a medium-sized data center of traditional IT service vendors while still making sufficient profit [12].

1.1 Motivation and problem statement

While cloud services promise cost reductions through flexible, on-demand usage of IT resources, using cloud services also entails risks related to, e.g., security, reliability and legal regulations since part of internal business processes are now supported through external IT resources and services [13]. Studies (e.g., [14][15][16][17][18]) suggest that these risks hinder the wide-spread adoption of cloud services and inhibit companies as well as public

institutions to fully take advantage of the economic benefits that cloud services offer. Thus, the obvious questions at this point are [19]:

- How can a customer control these risks, that is, how to unveil and assess potential risks and ensure that essential requirements are met?
- If a customer may choose among multiple cloud services for a desired purpose, how can the customer determine which one fits her requirements best?

One answer to the above questions is provided by *cloud service certification*, an approach which aims to demonstrate that a cloud service complies with a set of particular requirements [20] called *controls*. Controls are descriptions of measures aiming to modify risk [21] which can be obtained from, e.g., cloud-specific certificate catalogs such as Cloud Control Matrix (CCM) [22] upon which the Cloud Security Alliance (CSA) STAR certificate [23] is based or general standards such as ISO/IEC 27001:2013 [24]. Whether a cloud service meets a set of controls is determined by *evidence* which are any facts about an object that can be obtained through observation, for example, through tests [25]. If the cloud service satisfies specified controls, then a *certificate* is produced, stating *compliance*.

Leveraging this approach, the European Commission identified cloud service certification as one integral part of the European Cloud Strategy [11] which aims to foster adoption of cloud computing within the European ICT sector. In the course of implementing this strategy, the *Cloud Certification Schemes List (CCSL)*[1] has been compiled which consists of certification schemes potentially relevant to cloud customers.

While development of standards for cloud services and certification schemes is well under way, providing suitable methods to support the peculiarities of cloud service certification is subject to ongoing research. The core challenge of cloud service certification can be summarized as follows: When naively applying the concept of traditional certification to cloud services, a discrepancy surfaces because, traditionally, a certification 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 [26]. A cloud service, however, may change over time where the changes are hard to predict or detect by a cloud service customer [27]. These changes may lead to the cloud service not fulfilling one or more certificate's controls, thus rendering the certificate invalid. Therefore, the assumption of stability underlying traditional certification does not hold in context of cloud services. A popular example of a cloud service property which may change over time is the location of its components since migrating virtual components from one geographical location to another is a standard feature which cloud service providers such as Amazon Web Service (AWS) and Google CloudPlatform provide. Other examples include configuration changes of, e.g., security groups used to restrict access to cloud services, as well as changes of a cloud service's composition, e.g., temporarily adding new resources such as virtual machines. Cloud service certification thus requires a different approach capable of *continuously*, i.e., *automatically* and *repeatedly* detecting ongoing changes of a cloud service during operation and assessing their impact on satisfaction of a certificate's controls [28][29]. This implies that methods of *continuous cloud service certification* are not limited to simply repeating a defined task to assess a cloud service's properties over time but have to consider changes the service may undergo and self-adapt accordingly. Naturally, *continuous certification* is not to be understood strictly

[1] `https://resilience.enisa.europa.eu/cloud-computing-certification` [Accessed: 2018-12-13]

mathematically; no matter how sophisticated the method to check whether a cloud services complies with a set of controls, these results will always be – in a strict mathematical sense – produced by discrete tasks that occur at some point in time.

Aside from this service centered motivation, continuous certification is also explicitly required by some certification schemes themselves: For example, the US Federal Risk and Authorization Management Program (FedRAMP) [30], a government-wide program defining standards how to use cloud services, requires *continuous monitoring* of cloud services. However, FedRAMP's usage of the term *continuous* only refers to repeated but not necessarily automated checks of controls. Similarly, the *Cloud Computing Compliance Controls Catalogue (C5)* [31] issued by the German Federal Office for Information Security (BSI) advocates that audits of cloud services shall be renewed at least every 12 months. Going one step further, the influential guideline *Special Publication (SP) 800-37* [32] published by the National Institute of Technology and Standards (NIST) mandates using continuous monitoring techniques supported by automated means to enable a near real-time risk management. Also, the highest level of CSA STAR assessments called *CSA STAR Continuous Monitoring*[2] requires automated and repeated checks of controls.

Lastly, continuous certification may be implied on the level of an individual control's description, i.e., some controls themselves motivate automated and repeated checks. Consider, for example, *IVS-02 Infrastructure & Virtualization Security Change Detection* of CSA's CCM [22] which states that

> *"The provider shall ensure the integrity of all virtual machine images* ***at all times*** *[...]."*.

Other controls are less explicit but still imply that they can be continuously checked. One example is the *RB-03 Capacity management – data location* of BSI C5 [31] requiring that

> *"The cloud customer is able to determine the locations (city/country) of the data processing and storage including data backups."*.

Recent research has made important contributions to advance automated means supporting cloud service certification in the course of the EU-funded research project *CUMULUS*[3] which stands for *Certification Infrastructure for Multi-Layer Cloud Services*. They propose:

- *Monitoring-based certification* which uses monitoring data produced during productive operation to check controls (e.g., [28][33]);
- *Test-based certification* which checks controls through controlling some input to the cloud service, and evaluating the output, e.g., calling a cloud service's RESTful API and comparing responses with expected results (e.g., [34][35]); and
- *TPM-based certification* which builds on *Trusted Computing* in the form of Trusted Platform Modules (TPMs) [36] to provide proofs that the hardware layer of a cloud service's infrastructure is trustworthy (e.g., [37][38]).

All current approaches to cloud service certification – in particular those proposed within CUMULUS – have in common that they are *invasive* by design, that is, they require structural changes to the infrastructure used to operate the cloud service under certification (e.g.,

[2] `https://cloudsecurityalliance.org/star/continuous/` [Accessed: 2018-12-13]
[3] `https://cordis.europa.eu/project/rcn/105141_en.html` [Accessed: 2018-12-13]

installing monitoring agents on virtual machines). Furthermore, although current research presents means to design and implement automatic checks to support validation of controls, none of the current work incorporates the notion to explicitly execute *automated* and *repeated* checks which are able to *self-adapt* to changes of cloud-services over time. As a consequence, current research lacks a comprehensive discussion as well as methods on how to reason about sequences of results produced by continuous cloud service certification approaches. Lastly, current research efforts do not consider cloud service certification in presence of a fraudulent provider who is actively cheating on checks to create the appearance of compliance with a certificate's controls when, in fact, he is not.

Addressing these gaps, this thesis proposes a framework to guide the design and representation of minimally invasive tests to support continuous test-based certification of cloud services. This includes the introduction of universal metrics which can be computed on the basis of sequences of test results produced by any continuous tests. These universal metrics also allow us to evaluate and compare the accuracy and precision of continuous test results. Also, we introduce a model to reason about the behavior of a cloud service provider who only pretends to comply with the controls of a certificate. This model permits us to derive countermeasures through adapting tests provided by our framework.

1.2 Research challenges

This thesis centers around methods which aim to support continuous test-based cloud service certification. Hereafter, we describe the core research challenges which need to be overcome in order to develop these methods.

1.2.1 Challenge 1: Design of tests supporting continuous cloud certification

When intending to design tests to continuously produce evidence supporting validation of a certificate's controls, then an intuitive *top-down* approach starts from the perspective of a control: First, each control is inspected to decide which evidence is needed to check if this control holds. If necessary evidence has been identified, then one or more tests to continuously produce this evidence can be designed. This top-down approach has a major drawback: There are controls which do not manifest on the implementation level of a cloud service but target organizational processes and legal frameworks, involving, e.g., interviewing personnel of the cloud service provider or manual review processes [39]. Consider, for example, control *HRS-06* of CSA's Cloud Control Matrix (CCM) [22]:

> *"Requirements for non-disclosure or confidentiality agreements reflecting the organization's needs for the protection of data and operational details shall be identified, documented, and reviewed at planned intervals.".*

It is obvious that designing tests which supply evidence to check this control is a hard problem – if possible at all – since these test would have to comprehend the meaning of documents and assessing these in context of a certain history and needs of an organization.

Therefore, a different and preferable approach is to design tests *bottom-up*, i.e., starts from the perspective of the cloud service under test: First, we identify which technical *properties* of a cloud service we can evaluate using tests. For example, consider the property *availability*: A test can be designed which automatically and repeatedly checks whether the web server component of a Software-as-a-Service (SaaS) application is correctly and timely

responding to HTTP requests. Second, we have to identify those controls for which these test results can serve as evidence. Rigorously applying the latter step, however, has some fundamental limitations. In order to illustrate these limitations, consider the control *IVS-6* of the CSA's Cloud Control Matrix (CCM) [22]:

> *"Network environments and virtual instances shall be designed and configured to restrict and monitor traffic between trusted and untrusted connections. These configurations shall be reviewed at least annually, and supported by a documented justification for use for all allowed services, protocols, and ports, and by compensating controls.".*

Although the technical properties of a cloud service underlying this control can be easily derived, a fundamental problem surfaces which we refer to as the *semantic gap*: This control's generic description allows for multiple interpretations leading to various suitable evidence and corresponding candidate designs for continuous tests. For example, it is ambiguous what may indicate *untrusted connections* on implementation level or which are *allowed services, protocols, and ports*. Naturally, this level of abstraction further increases when considering controls of standards and guidelines which are not specific to cloud services, e.g., ISO-27001:2013 [24].

The reason that these controls are rather generic is that they are to be applied to a wide range of cloud services, regardless of type, particular service instances or their implementation. This level of abstraction may be intended by the authors of a certificate's controls to permit wide-spread application of the certificate. Yet this level of abstraction removes constraints on feasible interpretations necessary to derive the underlying cloud service properties. As a result, the decision which test design produces evidence suitable to validate a control becomes somewhat arbitrary, allowing for alternative tests. It is important to note that bridging the semantic gap between controls and tests, i.e., providing a method to rigorously derive tests from control descriptions is *not* in scope of this thesis.

To summarize: There is a *semantic gap* between the results produced by continuous tests and the controls for which these tests aim to provide evidence for. As of today, what constitutes suitable evidence to validate a control is left to informal interpretation of the human auditor. If we want to provide evidence using tests, then these informal interpretations of controls have to be made explicit. Using the terminology of Maibaum and Wassyng [40], continuous tests thus should be formulated as an *agreed-upon objective function* allowing objective, repeatable as well as predictable production of evidence. To that end, a framework to rigorously design continuous tests which allow reasoning about technical properties of cloud services is needed where the results of these tests can serve as evidence produced in an objective manner. This framework, however, does *not* solve the problem how to *agree upon* the objective function, that is, how to rigorously derive tests from descriptions of controls.

1.2.2 Challenge 2: Definition of continuous tests

A certificate demonstrates that a cloud service satisfies a set of controls. The goal of that demonstration is to increase a customer's trust towards a cloud service and enable comparability between different cloud services [19]. When using tests to produce evidence to validate at least part of a certificate's controls, it is vital to ensure that the *definition of how evidence is produced* is *unambiguous*. Only unambiguously defined tests allow us to

rigorously compare evidence, that is, the test results produced by different tests. Consider, for example, the control *RB-02 Capacity management – monitoring* of the Cloud Computing Compliance Controls Catalogue [31] 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 we aim to compare the *availability* of two SaaS applications where both SaaS providers have defined the identical service level agreement (SLA) regarding availability (e.g., 99.9999% per year) and both providers claim to fulfill control RB-02 of BSI C5. In order to check whether their claims hold, each application is continuously tested to detect potential outages, i.e., periods where the service is unavailable. Yet the continuous test used by each application differ: One service's availability is tested by simply pinging its endpoint, i.e., measures the time delta between sending ICMP Echo Request and receiving ICMP Time Reply packets. Measured round trip times are then compared with expected ones. The other application is tested by issuing specially crafted calls to its RESTful API and comparing the returned JSON object with expected object using some proprietary comparison function. It is obvious that two different continuous tests produce evidence that differ in semantics. We use this evidence to support evaluation whether the respective SaaS application complies with the above control. As a consequence, the conclusions drawn based on the differing evidence produced for the two example SaaS applications cannot be directly compared.

The above example illustrates that unambiguous definition of evidence production is needed to provide comparability of results generated by tests. Thus, an approach has to be developed that lets us define continuous tests in a rigorous way. These *continuous test definitions* have to be *complete*, that is, adhere to the framework's concepts to design continuous tests (see *Research Challenge 1*) and contain all information required to configure specific test implementations. At the same time, these definitions have to be general representations of continuous tests which are agnostic to specific test implementations, thereby enabling comparability.

1.2.3 Challenge 3: Accuracy and precision of continuous test results

Inaccurate tests undermine both cloud service provider's and customer's trust: On the one hand, test results that *incorrectly* indicate satisfaction of a control erode customer's trust. On the other hand, cloud service providers will dispute test results that incorrectly suggest controls are not fulfilled. Therefore, it is essential to evaluate the accuracy and precision of continuous test results, that is, how close are produced test results to their true values?

Consider the control *TVM-02: Threat and Vulnerability Management Vulnerability / Patch Management* of CCM [22] whose first part reads

> *"Policies and procedures shall be established, and supporting processes and technical measures implemented, for timely detection of vulnerabilities within organizationally-owned or managed applications, infrastructure network and*

> *system components (e.g., network vulnerability assessment, penetration testing) [...]."*.

One possibility to produce evidence supporting validation of this control is a test which executes a vulnerability scanner every ten minutes and checks whether no vulnerability is found. The question is now whether this test makes mistakes by, e.g., incorrectly indicates that the cloud service under test has no vulnerabilities while it actually has. In this case, it is unclear to what extent the produced evidence can be used to check the control. Does, e.g., the vulnerability scanner always miss detecting a particular vulnerability or merely occasionally?

Let's now assume that the test for security vulnerabilities only produces correct results. When inspecting control *TVM-02*, it becomes apparent that it not only requires to detect security vulnerabilities but also demands remedy within specific period of time. In context of such temporal constraints, further errors may occur when – based on the continuous test – estimating the duration of detected vulnerabilities.

An approach is needed to evaluate the accuracy of test results which are produced continuously. This approach has to allow for inference of conclusions about the general accuracy of continuous tests' results. Furthermore, this method needs to permit comparison between the accuracy of alternative tests which can be used to produce evidence to validate a particular control.

A resulting challenge when evaluating the accuracy of continuous tests lies in establishing the ground truth to which test results are compared. To that end, an approach is needed which allows manipulating properties of a cloud service under test resulting in violation of controls which a continuous test aims to detect. For example, a control violation may consists of starting and stopping virtual machines to violate resource availability, publicly expose sensitive interfaces to violate secure configurations, or restrict available network bandwidth to violate quality of service.

1.2.4 Challenge 4: Trustworthy continuous test results

Through demonstrating compliance with a set of controls and thus obtaining a certificate, cloud service providers hope for competitive advantages, e.g., by increasing customer retention and attracting new customers [19]. However, measuring customers' preferences as well as general effects of cloud service certification are subject to ongoing research (e.g., [41][42][43]), thus making it difficult for cloud providers to predict the expected additional revenue generated through obtaining a particular certificate.

In contrast, complying with controls of a certificate can lead to an increase in costs of a cloud service provider which are straightforward to quantify. Consider, for example, operational efforts involved when guaranteeing high service availability and reliability which can be directly translated into costs.

The contrast of certification's uncertain benefits in the mid and long term versus certain increase in cost of service provisioning in the short term can create an incentive for a cloud service provider to cheat: A *fraudulent* provider may only pretend to satisfy the controls of a certificate in order to reduce his costs. As an example, consider the control *UP-02 Jurisdiction and data storage, processing and backup locations* of BSI C5 [31] whose second part states that

> *"[...] Data of the cloud customer shall only be processed, stored and backed up outside the contractually agreed locations only with the prior express written consent of the cloud customer."*.

Let's assume that a cloud service customer uses Infrastructure-as-a-Service (IaaS) supplied by a public cloud provider to host a social network platform where registered users can connect, chat, share pictures and so forth. The certificate for this IaaS provider states that all resources are hosted in compliance with control *UP-02* of BSI C5. In our example, we assume that the IaaS provider contractually agreed to only use virtual machines, persistent storage and related infrastructure services which are hosted within the European Union (EU).

However, our IaaS provider is fraudulent since he only complies with control *UP-02* during the day, i.e., from 6am to 11pm (UTC) while – in order to save costs – during the night, i.e., from 11pm to 6am (UTC), the provider uses resources hosted at data centers located in geographical regions outside of EU. Naturally, this migration is transparent to the cloud service customer, that is, the customer cannot discern which geographical location is used to provide the cloud service. Thus, the provider intentionally violates the control and only pretends to comply with the certificate's controls.

If we use tests to continuously produce evidence supporting validation of certificates' controls, then not trusting the cloud service provider implies that we cannot unconditionally trust test results. Under these circumstances, it is vital to provide an approach how to adapt continuous tests to produce trustworthy test results, even when faced with a fraudulent provider.

1.3 Contributions

In this thesis, we make the following contributions:

- **A framework to design continuous tests** We introduce a framework to guide the design of tests to support continuous test-based cloud service certification and describe an example implementation called *Clouditor*. Using this framework, minimally invasive tests can be crafted that produce evidence to validate fulfillment of a certificate's controls. The framework also supports self-adaptive testing to be able to react to changing environment conditions of the test as well as to consider previously produced test results. Further, we propose four universal test metrics which can be used to reason about sequences of results produced by any continuous test designed according to the framework. Lastly, existing approaches to test-based certification can leverage our framework to execute test repeatedly and reason about sequences of produced test results.

- **Example continuous test scenarios** We present five test scenarios to show how our framework supports continuous test-based cloud service certification according to controls derived from BSI C5, CSA STAR, and ISO/IEC 27001:2013. Within these scenarios, we show how to continuously test *availability*, *location*, *secure communication configuration*, *secure interface configuration*, and *user input validation*. Further, since a scenario-driven approach is limited in demonstrating applicability of our framework, we identify four general characteristics to describe continuous test scenarios allowing us to reason about the applicability of our framework beyond the scope of the scenarios.

- **A domain-specific language to define continuous tests** We do *not* mandate a specific architecture to implement tests according to our framework. Instead, in order to guide development of tests according to our framework, we make use of formal languages

to define a domain-specific language called *ConTest* with which continuous tests can be rigorously described. ConTest is a descriptive language which, on the one hand, provides a general representation of continuous tests which is agnostic to a specific implementation of our framework's building blocks. On the other hand, ConTest serves as a starting point to generate concrete configurations for any specific implementation of a continuous test. As a result, having a developer provide a code generator to translate from ConTest to an implementation-specific test configuration language ensures that the configuration of the test implementation adheres to the building blocks of our framework.

- **A method to evaluate accuracy and precision of continuous test results** We introduce a method to experimentally evaluate the accuracy and precision of results produced by continuous tests. This method allows us to compare alternative tests as well as alternative test configurations. At the core of this method are *accuracy and precision measures* which are based on the universal metrics proposed as part of our framework to design continuous tests. Thus our method can be used to evaluate the accuracy and precision of any test which is designed using the building blocks of our framework.

  In order to establish the ground truth required to evaluate the accuracy of a continuous test, we presented the notion of *control violations* which manipulate a property of a cloud service under test such that the service does not fulfill one or more controls of a certificate. Furthermore, we explain how randomization of control violation events allow to generalize statements about a continuous test's accuracy. Based on three example scenarios, we show how the accuracy and precision measures allow us to compare alternative tests as well as alternative test configurations to select the most suited one.

- **Test-based certification of adversarial cloud service providers** We present a model of an adversarial cloud service provider who only pretends to comply with a set of controls. This type of fraudulent provider is referred to as *opportunistic cloud service provider* who only cheats if he can be sure with a certain probability that he is not caught. In order to be able to model and reason about the behavior of an opportunistic provider, we draw on the idea of *covert adversaries* introduced by Lindell and Aumann [44] and combine it with a modeling technique known as Labeled Transition Systems (LTS). Based on our model, we then show that randomization of tests can reduce the willingness of the opportunistic provider to cheat. Further, we explain how our framework to design tests supports randomization of tests on different levels.

1.4 Organization of this thesis

The remainder of this thesis is organized as follows:

- Chapter 2 provides an overview of focal concepts of the three fields relevant to this thesis: *Cloud services*, *testing* as well as *certification*. Beyond the introduction of these concepts, we point out how and to what extent these fields relate to this thesis.

- Chapter 3 presents research efforts related to this thesis, points out gaps of the status quo and explains how this thesis addresses missing parts. To that end, we provide an in-depth

discussion of current approaches to test-based cloud service certification, including other frameworks explicitly developed to support cloud certification, specialized methods aiming at checking specific cloud service properties as well as other approaches whose primary goal is to compare performance of cloud services but which still can be used to support test-based certification. Further, we also provide an overview of monitoring-based and TPM-based cloud service certification since these depict the major other approaches to cloud service certification. Lastly, we also shed light on *Testing as a Service (TaaS)*, i.e., using cloud services to execute tests since the implementation of our continuous test scenarios in Chapter 5 follow the notion of TaaS.

- Chapter 4 introduces our framework to design tests which aim at continuously producing evidence to validate satisfaction of controls of certificates. We start with laying out the requirements the framework has to meet and then describe its main building blocks, i.e., *test cases*, *test suites*, *workflow*, *test metrics* and *preconditions*. We explain how these building blocks correspond with the initially defined requirements and, finally, present an example implementation of the framework called *Clouditor*.

- Chapter 5 presents five example test scenarios designed and implemented using the framework described in Chapter 4. The goal of these test scenarios is to demonstrate how our framework can support continuous test-based certification of cloud services. We begin with defining general characteristics of the test scenarios which we then apply to each selected scenario. Thus we are not limited to the example five test scenarios but can draw conclusions about the applicability of our framework beyond these test scenarios. Thereafter, we describe continuous tests of *availability*, *location*, *secure communication configuration*, *secure interface configuration*, and *user input validation* of cloud services.

- Chapter 6 proposes the domain-specific language *ConTest* which permits rigorous definition of continuous tests while being agnostic to specific test implementations. To that end, we first outline basic concepts of formal languages and engineering of domain-specific languages. Then we present the design as well as implementation of ConTest and show how ConTest can be used to generate configurations for specific test implementations which are used to configure Clouditor.

- Chapter 7 introduces a method to experimentally evaluate the accuracy and precision of continuous test results. After having defined the terms accuracy and precision in the context of continuous test-based cloud certification, we describe how to violate cloud services' properties in order for the service not to fulfill controls. Then we introduce accuracy and precision measures to evaluate continuous test results, including the inference of conclusions about the general accuracy and precision of test results. Finally, we present experimental results of applying our method to evaluate and compare results of tests which aim to support validation of controls related to availability, security and reliability.

- Chapter 8 presents the notion of the *opportunistic cloud service provider*, a special type of fraudulent cloud service provider who only pretends to comply with controls while he is actually not. We begin with introducing *covert adversaries* and *Labeled Transition Systems (LTS)*, the two main concepts we use to model and reason about the behavior of an opportunistic cloud provider. Thereafter, we describe how we use

LTS to model opportunistic cloud service providers and show how randomization of tests can reduce the willingness of opportunistic providers to deceive. Finally, we point out how our framework to design continuous tests presented in Chapter 4 supports randomization of test parameters as one possible countermeasures against opportunistic cloud service providers.

- Chapter 9 concludes this thesis by summarizing and discussing its main results as well as pointing out directions for future work.

Chapter 2

Background

This chapter introduces concepts and terminology of three fields – *cloud computing*, *testing* and *certification* – which are required to understand the remaining chapters of this thesis. Further, we explain the scope and relevance of these three subjects in context of this thesis.

The next section outlines the notion of cloud computing, including different service and deployment models. We omit an in-depth historical review of the origins of cloud computing, considering utility computing, grid computing and the like since such contextualization is already extensively covered in the literature, e.g., [45][46]. Section 2.2 presents basic concepts of software testing where particular emphasis lies on security and conformance testing. Lastly, in Section 2.3, we introduce the basic ideas behind software, service and cloud service certification.

2.1 Cloud computing and cloud services

In their influential paper *Above the Clouds: A Berkely View of Cloud Computing* published in 2010, Armbrust et al. [12] pinpoint three key characteristics of cloud computing:

1. The illusion of infinite resources which are available on demand,

2. absence of an up-front investment for hardware, and

3. pay per use for resource for only as long as they are needed.

While the term *cloud computing* is heavily overloaded and there are numerous attempts to define it (e.g., [47][48][49]), Armbrust's perspective provides a first distinction between cloud computing and regular outsourcing of hardware and software: In [12], they argue that

> *"[...] the construction and operation of extremely large-scale, commodity-computer datacenters at low-cost locations was the key necessary enabler of Cloud Computing, for they uncovered the factors of 5 to 7 decrease in cost of electricity, network bandwidth, operations, software, and hardware available at these very large economies. These factors, combined with statistical multiplexing to increase utilization compared a private cloud, meant that cloud computing could offer services below the costs of a medium-sized datacenter and yet still make a good profit."*.

Aside from this mainly economic characterization, an important question is: How can cloud computing and cloud services be described from a technical point of view? Answering this question, multiple proposals are available which aim at establishing a reference architecture for cloud services, e.g., [50][51][52][53][54]. However, as of writing of the thesis, the canonical answer to this question is provided by the National Institute of Standard and Technology (NIST) whose definition of cloud computing is published in [10]. While this definition is still fairly abstract, it nevertheless provides a starting point to define some terminology which we are going to use extensively within this thesis. The following two sections will define the terms *service model* and *deployment model* on the basis NIST's understanding, supplementing it with examples of currently popular cloud service implementations.

2.1.1 Cloud service models

According to Vaquero et al. [55], *cloud service providers* make cloud services available to *cloud service customers* using IP-based protocols. As a result, cloud service customers do not have to maintain and run the underlying infrastructure necessary to host the service. Further, Vaquero et al. [55] introduce three types of services, i.e., Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) which are later adopted by Mell and Grance [10] proposing the NIST's definition of Cloud Computing. These service models are detailed hereafter.

Infrastructure-as-a-Service (IaaS) In this service model, the cloud service customer is provided basic computing resources such as processing, storage and network. IaaS offers the highest degree of freedom since any software or application can be assembled and run on top of IaaS [10]. As already pointed out above, the key technology at this level is virtualization which allows abstracting from concrete physical computing resources and thus divide, (re-)allocate and (re-)size basic computing resources in an ad hoc manner according to service customers' needs [55]. However, while the service customer controls, e.g., setup and configuration of virtual machines, attached network interfaces and volumes, the customer does not have to manage the underlying infrastructure required to provide IaaS [10].

According to the Cloud Security Alliance (CSA) [56], Amazon Web Service (AWS)[4] is the leading IaaS provider as of 2017 with *Amazon Elastic Compute Cloud (EC2)*[5] being their core product, offering virtual machines with various configurations. Other IaaS provider with relevant market share are Microsoft Azure[6], Google CloudPlatfrom[7] and IBM Cloud[8].

Platform-as-a-Service (PaaS) PaaS is considered to deliver services on a level which has a higher degree of abstraction than IaaS: Instead of providing only fundamental computing resources as a service, PaaS providers give means to service customers to build and run applications within their cloud infrastructure [55]. In contrast to IaaS, service customers are bound to the tools, languages, libraries and so forth which the PaaS provider supports [10]. The advantage of this service model, however, lies in transparent resource management (for

[4] `https://aws.amazon.com/` [Accessed: 2018-12-13]
[5] `https://aws.amazon.com/ec2/` [Accessed: 2018-12-13]
[6] `https://cloud.microsoft.com/` [Accessed: 2018-12-13]
[7] `https://cloud.google.com/` [Accessed: 2018-12-13]
[8] `https://www.ibm.com/cloud-computing/` [Accessed: 2018-12-13]

the service customer) which is needed to run applications (e.g., workload distribution, up and down scaling of resources etc.).

A popular example for a PaaS offering is the Google App Engine[9] which allows service customers to develop and deploy application written in Node.js, Java, Ruby, C#, Go, Python, and PHP. AWS offers a comparable product called *Elastic Beanstalk*[10] while Salesforce provides a PaaS named *Force.com*[11] which is specialized to support development of applications extending their core product in the form of, e.g., mobile applications.

Software-as-a-Service (SaaS) From a service customer's point of view, using Software-as-a-Service (SaaS) applications can be understood as an alternative to running applications locally, i.e., at the customer's site [55]. SaaS applications are deployed on remote infrastructures which are accessible through interfaces such as browsers or stand-alone clients which are installed on the local workstation of the service customer. Here, the customers' control of the cloud service is confined to configuring user-specific application settings [10]. Providing SaaS applications heavily involves web application technologies, such as JavaScript, JSON, HTML and CSS.

Popular examples for SaaS applications are Google *G Suite* Apps[12] and Microsoft 365[13], both providing office applications such as email, docs and calendar as a service. Other examples include the customer relationship management (CRM) application provided as a service by Salesforce[14] as well as SAP Business ByDesign[15] offering an enterprise resource planning (ERP) application as a service.

2.1.2 Deployment models

The NIST definition of cloud computing [10] further distinguishes whether a particular cloud services is exclusively used by a single organization or shared among more than one organization. The former case is referred to as *private cloud services*. Note that the infrastructure used to implement private cloud services does not necessarily have to be managed by or deployed in the data center of the organization using them. A popular, free and open source platform which can be used to provide private IaaS is OpenStack[16]. The prime example of a *public* cloud service provider is AWS whose services are – with a few exceptions – accessible for the general public. A public cloud service provider does not necessarily need to own and maintain the physical infrastructure used to provide the public cloud services either. Other subcontracted companies can be in charge of housing and facility management of the data centers. Lastly, NIST distinguishes the deployment models *community cloud services* and *hybrid cloud services*. The former simply refers to services which are shared among a closed group of organizations which have similar requirements towards the cloud service. The latter can be perceived as a meta-type as it describes service combinations of private, public or community cloud services.

---

[9] `https://cloud.google.com/appengine/` [Accessed: 2018-12-13]

[10] `https://aws.amazon.com/documentation/elastic-beanstalk/` [Accessed: 2018-12-13]

[11] `https://www.salesforce.com/products/platform/products/force/` [Accessed: 2018-12-13]

[12] `https://gsuite.google.com/` [Accessed: 2018-12-13]

[13] `https://products.office.com/en/business/explore-office-365-for-business` [Accessed: 2018-12-13]

[14] `https://www.salesforce.com/crm/` [Accessed: 2018-12-13]

[15] `https://www.sap.com/india/products/business-bydesign.html` [Accessed: 2018-12-13]

[16] `https://www.openstack.org/` [Accessed: 2018-12-13]

2.1.3 Cloud services in context of this thesis

We consider all three major types of cloud service models in this thesis: For example, supporting all three cloud service models is one key requirement driving the development of our framework to support continuous cloud certification (see Section 4.2.4). Also, in Section 5.1, we discuss implication of different services models in context of our framework as well as show through the implementation of selected test scenarios how our framework supports testing of all three services models.

With regard to deployment models, we focus on public cloud services while private, community and hybrid cloud services play little to no role in this thesis. For example, one central requirement of our framework to support continuous tests (see Section 4.2.3) consists of flexible integration with existing cloud services. In this context, public cloud services are particularly important since we have to assume that the public cloud provider will try to minimize integration efforts, thus leading to the notion of *minimally invasive* testing. Further, public cloud services provided by AWS play a crucial role within the example test scenario to validate the location of a cloud service component (see Section 5.3). Lastly, public cloud service providers play a major role in Chapter 8 where we assume the cloud provider to be *opportunistic* which is a special type of adversarial behavior.

Note that for budget reasons, we conducted the majority of experiments presented in this thesis (e.g., Section 5.2.4, 5.4.4 or 5.5.4) using a cloud environment based on OpenStack. Although access to this environment is restricted to internal use only, thereby rendering it strictly speaking a private cloud, any experiment presented in thesis can easily be repeated using commercial public cloud services, e.g., AWS or Google Cloud Platform. The reason for this is that none of the experiments presented herein require privileged access to the underlying infrastructure which is used to operate and provide the cloud service.

2.2 Testing

This section introduces basic concepts of software testing, security testing as well as conformance testing.

2.2.1 Software testing

A standard definition of testing is provided by the *Standard Glossary of Terms used in Software Testing* [57] published by International Software Testing Qualifications Board (ISTQB). This glossary states that *testing* is

> *“the process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.”*.

Yet this definition rather delineates the goal which testing pursues. The IEEE Guide to the software engineering body of knowledge (SWEBOK) [58] offers a more descriptive definition:

> *"Software testing consists of dynamically verifying a program’s behavior on a finite set of test cases – suitably selected from the usually infinite domain of executions – against the specified expected behavior."*.

Utting and Legeard [1] decompose this definition by detailing the following key aspects:

- *Dynamic testing:* Refers to evaluating the behavior of a software or application through observing it during execution while *static* testing evaluates the software without execution by, e.g., manually reviewing the source code.
- *Finite set of test cases:* Exhaustive testing of software is usually either prohibitively expensive, takes too long, or is simply not practically feasible. The reason for this is that the possible input domain for a test function can be very large, possibly infinite. Thus some constraints on the set of possible test cases, e.g., maximum duration of a test have to be defined.
- *Selected test cases:* As we are bound to a finite set of test cases, selecting those test cases which are most likely to unveil faults of the tested software, e.g., security vulnerabilities, presents a key challenge. One complementary strategy here is *risk-based testing* where test cases are selected such that the risk for the software user is minimized [57]. This means that testing focuses on parts of the software which have been identified to most likely have some defects with considerable impact on, e.g., the data security of the software user [59].
- *Expected behavior:* After having executed a test, the observed behavior of the software under test needs to be interpreted to decide whether the test failed or passed. This mechanism is also known as the *test oracle*. In case of automated testing, automatically reasoning about the observed behavior is a necessity.

If an undesired behavior is observed, then this is referred to as a *failure* where the cause of this failure is called *fault* [60]. In short, we can summarize that testing is a method to detect faults [61]; however, it is crucial to understand that testing cannot show that a software does *not* possess any faults [62].

Types of dynamic testing Having understood the general definition of testing, we now describe different types of dynamic testing according to the three dimensions shown in Figure 2.1 [1][2]:

- *Scope* delineates at what level the system under test (SUT) is evaluated. *Unit testing* aims at testing a single unit of the SUT at a time, e.g., methods implemented as part of particular class [60]. *Component testing* focuses on parts within the SUT, e.g., a set of classes, deliberately neglecting other parts of the SUT which is usually achieved by manipulating the SUT through specialized, internal code interfaces [63]. *Integration testing* aims at discovering faults in interfaces as well as between components interacting [57]. Yet integration testing still only considers subsystems of the SUT while with *system testing* the entire system is under test, including all components and subsystems, in order to evaluate its adherence to specified requirements [60].
- *Accessibility* describes the information available to design a test. *Black box testing* (also known as *specification-based* or *behavioral testing* [57]) assumes that no information about the internal structure of the SUT is available to the tester. Thus, the test design is driven by the inputs the SUT accepts as well as its output, i.e., its expected behavior as specified by the system requirements [1]. In *white box testing* (also known

as *structure-based* or *structural* testing), any required information about the inner workings of the SUT are available to the tester and thus tests can be designed using this information [57]. Further, the quality of structural testing activities is often evaluated using measures of *code coverage* such as *branch coverage* which describes the ratio between exercised branches and all branches defined in the code [64]. However, while measures of code coverage support detection of untested parts of source code, they do not allow to make statements about the quality of the conducted tests [65].

- *Objective* refers to the characteristics of the SUT we aim to test. *Functional testing* checks whether the SUT meets specified functional requirements where internal structure of the SUT is neglected, thereby rendering it a black box test method [60]. *Non-functional testing* aims at validating non-functional requirements such as robustness, performance, reliability, and security requirements [1].

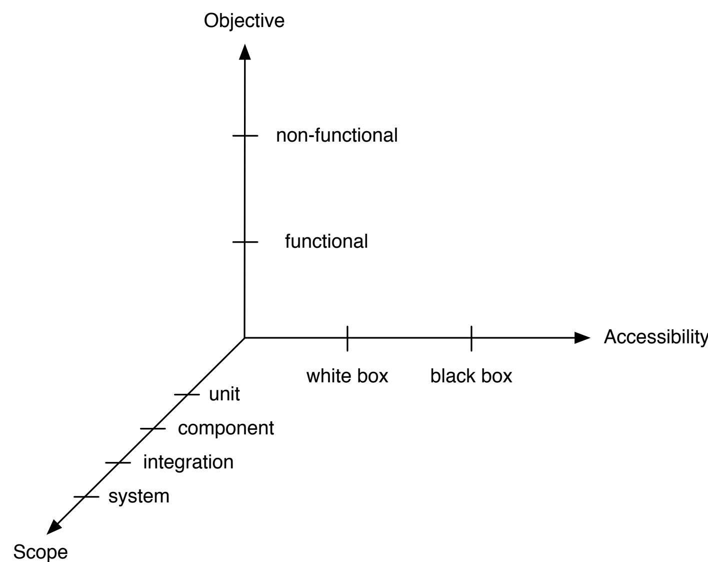


Figure 2.1: Dimensions of testing types (based on Figures from [1][2])

Test process Conducting a test requires additional steps to prepare the test and follow up on the results. According to [57], the test process consists of the following steps:

1. *Planning* includes all activities to create and maintain a test plan, i.e., a document which describes scope, approach, resources and schedule of planned test activities as well as when to stop testing, i.e., exit criteria. According to Felderer et al. [2], such exit criteria can be based on *coverage criteria* describing the desired *test coverage*, that is, the degree to which a set of tests addresses all requirements defined for the software under test [60]. Thus coverage criteria can be understood as characteristics of a test or a set of tests which specify the extent of testing considered sufficient to demonstrate the SUT complies with all requirements [59].

2. *Design* translates test objectives defined by the test plan into descriptions of executable tests. Depending on the accessibility of information about the software to be tested,

tests can either be specification-based (i.e., black box tests) or structure-based (i.e., white box).

3. *Implementation* takes the specification provided by the test design step and creates test procedures, including needed test data, as well as preparing frameworks supporting automated test execution (also known as *test harnesses*).

4. *Reporting* consists of the collection and analysis of test results.

5. *Closure activities* finalizes the test process through consolidating all results obtained after completion of all test activities defined by the test plan.

Continuous testing Continuous tests are tests which are executed automatically and repeatedly [66]. Naturally, *continuously* is not meant strictly mathematically since executing tests are always – in a strict mathematical sense – discrete tasks that occur at some point time. What the term *continuous testing* does suggest, however, is that tests are conducted very frequently, e.g., on every change made to the source code of an application. Continuous testing implies automated testing, thus not all types of testing can be conducted continuously, e.g., code reviews (for further information see paragraph *Static security testing techniques* in Section 2.2.2).

Continuous testing can support *regression testing* (e.g., Saff and Ernst [66, 67, 68]), that is, retesting a software after is has been modified [57]. In case of continuous white box testing, changes made to the code base can be observed and thus used as triggers to retest the modified software. This kind of incremental building and testing of software is described by the term *continuous integration* [69] which works as follows: Every change a developer makes the code base is committed to a central source code management system such as SVN[17], GIT[18] or TFVC[19] which is usually deployed on a different server. On every commit, the source code management system triggers the *build system* implemented by tools such as *Jenkins*[20], *Hudson*[21], *Bamboo*[22] or *Travis CI*[23]. Those tools control compilation and packaging of the software and trigger, e.g., unit tests after every build of the software.

In the case of continuous black box testing, per definition, no changes to internal mechanisms of the software can be observed externally. Therefore, it is not possible to trigger the execution of a black box test based on changes applied to the code. As a result, different strategies have to used to identify the point in time when to execute a black box test. Basic strategies are choosing time of execution randomly or triggering the test based on static intervals, for example, run vulnerability scans of an application every night. More advanced strategies may use other externally observable events which indicate that a change has occurred, e.g., the web server version of a web application changed.

Testing for validation versus testing for verification As already described above, a *test oracle* is the mechanism indicating whether the observed behavior of the SUT conforms with

[17]`https://subversion.apache.org/` [Accessed: 2018-12-13]
[18]`https://about.gitlab.com/` [Accessed: 2018-12-13]
[19]`https://www.visualstudio.com/team-services/tfvc/` [Accessed: 2018-12-13]
[20]`https://jenkins.io/` [Accessed: 2018-12-13]
[21]`http://hudson-ci.org/` [Accessed: 2018-12-13]
[22]`https://www.atlassian.com/software/bamboo` [Accessed: 2018-12-13]
[23]`https://travis-ci.org/` [Accessed: 2018-12-13]

the expected behavior. Depending on what source is used to derive the *expected behavior*, we can distinguish between *testing for validation* and *testing for verification*. Based on ISO/IEC/IEEE 24765:2017 [60], we can describe the difference between validation and verification as follows: *Validation* aims at confirming that requirements defined for an application are fulfilled based on evidence, thereby assuring that the application meets the needs of the users. In contrast, *verification* refers to the process which evaluates if the outcome of a step of an application's development process satisfies the conditions defined at the beginning of that step. Drawing on these descriptions, we can state that *testing for validation* means to check whether a SUT conforms with customer's requirements. Further, *testing for verification* delineates testing a system or application against (possibly formalized) specifications.

2.2.2 Security testing

Security testing refers to dynamic and static testing techniques with the objective to validate whether security requirements are satisfied; designing security tests can either build on available information about the internal mechanisms of the SUT (white box) or solely be derived from externally observable behavior of the SUT (black box) [70]. Security requirements can be related to *security properties* of SUT, e.g., *availability*, *confidentiality*, and *integrity*. Thus we can state that security testing aims at establishing whether the implementation of the SUT has a set of desired security properties [2].

Alexander [71] introduced the notion of positive and negative security requirements by proposing *misuse cases*, i.e., the malicious use of a system driven by an attacker's intent. Zech et al. [72] distinguish these two types of security requirements as follows: *Positive* security requirements describe some desired, expected behavior of a system, e.g., a browser shall only retrieve and interact with SaaS applications via HTTPS, a SaaS application's access management component shall always require two factor authentication to grant users access etc. *Negative* security requirements define behavior of a system that is *not* desired, i.e., actions that the system should not grant or execute, e.g., hostile cloud service customers should not be able to access, manipulate or disrupt the services of other customers.

The notion of positive and negative security requirements leads to two types of security testing introduced by Tian et al. [73]: *Security functional testing* and *security vulnerability testing*. While the former focus on testing the correctness and completeness of security mechanisms the SUT implements, the latter attempts to detect unintended side effects which lead to vulnerabilities. This distinction is also made by Thompson [74] who compares the intended behavior of a system with its implemented behavior. Thompson' key insight is that most vulnerabilities stem from unwanted side effects of a system's implementation.

Yet, as Felderer et al. [2] point out, security vulnerability testing is a hard problem: Recall that in the previous section, we stated that testing serves to detect faults. Strictly applying this definition to security vulnerability testing would mean that security vulnerability testing had to detect the absence of some additional behavior caused by unwanted side effects of a system's implementation. This is a hard problem because it implies to consider all possible states of the system which in practice is either undesired because it takes too long or is computationally too expensive (i.e., time and space) or it is not feasible at all. The obvious workaround is to focus on violating a security property (and thus the related security requirement), i.e., demonstrate through testing that a security property does *not* hold. The most popular security testing techniques following this notion is *penetration testing*.

Static security testing techniques These testing techniques do not require execution of the application. Manual *code review* is a technique where a security expert inspects the source code of an application under test by reading it in order to detect faults [75]. Since such manual code reviews are time-consuming and thus expensive tasks, supporting techniques have been developed [76]. Tools implementing these techniques are usually referred to as *static application security testing (SAST)* tools which reduce the manual effort required for analysis, thereby allowing to inspect larger code bases within practically relevant time bounds. SAST tools conduct syntactic checks using regular expressions to detect, e.g., insecure function calls [77] which, for example, could lead to *format bugs* [78]. Also, SAST tools execute *semantic checks* where program semantics are derived from control flow graphs which serve as an abstracted model of a program's state [79]. Such semantic analysis allows detecting, e.g., whether user input supplied to a web application is being sanitized before processed by the application to prevent SQL injection attacks [80].

Dynamic security testing techniques These testing techniques are applied while the SUT is running. *Penetration testing* is a technique where the penetration tester usually only has very little information about the SUT and tries to mimic the actions of real adversary in order to exploit potential vulnerabilities of the SUT [81]. In order to identify potentially exploitable vulnerabilities of the SUT, penetration testers make use of different tools, e.g., *vulnerability scanners* which automatically detect vulnerabilities of a target system by executing known attack patterns [2]. Penetration testing is usually conducted at the end of the software development lifecycle, i.e., just before the release and deployment of the system or application [82].

Another dynamic security testing technique is *fuzzing* which builds on a notion already introduced by Miller [83] in 1990. In its basic variant, this technique provides randomized inputs to the SUT with the objective of detecting vulnerabilities of the SUT, e.g., by disrupting its execution or forcing it into a vulnerable state. Advanced fuzzing techniques use more sophisticated means to generate the input to provide to the SUT, e.g., *mutation-based* and *generation-based* fuzzing [84][85][86].

The last technique we will consider in this outline is referred to as *dynamic taint analysis* or *dynamic information flow analysis*. This technique works as follows [87]: Input data from an untrusted source is tainted (i.e., marked in a defined way), e.g., form fields of a web application, and then the propagation of that tainted data within the application is tracked. It is then tested whether this tainted data is used in a malicious manner, for example, tainted data is used to format strings which may point to a format string vulnerability. Dynamic taint analysis can be used to detect vulnerabilities resulting from, e.g., buffer overflows (e.g., [87]), use of format strings (e.g., [88]) as well as SQL injections (e.g., [89]).

2.2.3 Conformance testing

In [90], the National Institute of Standards and Technology (NIST) points out that *conformance testing* can be understood as a meta-category of testing which may include tests with functional as well as non-functional objectives. However, its primary focus lies on validating whether a software meets a set of requirements of a standard or specification, which are usually defined in the form a *conformance clause* as part of a standard or specification. Without such a conformance clause, no conformance testing can be conducted. Lastly, results of conformance testing are *not* suited to compare two alternative software artifacts, they solely

demonstrate conformance with some standard or specification.

In the same article on conformance testing [90], NIST also outlines how to establish a *conformity assessment program* for standards that do not have a conformance clause. Such a program includes:

- The standard or specification to establish conformance with,
- the conformance clause, i.e., specification of how test conformance,
- tools and other procedures to conduct conformity assessment, and
- an entity to conduct the tests.

Finally, the NIST distinguishes between three different types of conformance testing:

- *Exhaustive testing:* Validate conformance with a standard by exploring any possible state of the system through testing. Naturally, for most applications, exhaustive testing is either prohibitively expensive or not practically feasible at all.
- *Thorough testing:* Here validation of a software artifact conforming with a standard is not exhaustive but bound to some range, e.g., not any possible combination of input values is used for testing but only a few selected ones, usually specified as part of the conformance clause.
- *Identification testing:* Compared to exhaustive and thorough testing, this type of conformance testing is least complete. Only focal properties of the software artifact are tested, providing only minimal input to the software under test.

2.2.4 Testing in context of this thesis

Testing techniques used within this thesis belong to the class of dynamic testing techniques since we test cloud services at runtime, that is, during their productive operation. Furthermore, we do not assume to have any information about the internal mechanisms of the cloud service under test. Therefore, our test designs follow the black box notion, i.e., the test designs are driven by the expected behavior which are defined by the controls of a certificate (for more detail see the next section) and which can be observed through interaction with the service. Further, our framework executes tests automatically and repeatedly, thereby rendering it continuous black box tests.

The scope of testing in this thesis focuses on component testing, that is, we test components of a cloud service such as virtual machines (see Section 5.2 and 5.3), web servers (see Section 5.4), firewalls (see Section 5.5), as well as data bases (see Section 5.6).

The test objective that we pursue with our continuous tests is to check if properties of a cloud service hold and thus produce evidence that the service fulfills some controls. This means that we are testing for validation. To that end, we make use of dynamic security testing techniques such as vulnerability scanners (see Section 5.6). However, aside from security vulnerability testing, we also make use of security functional testing, e.g., to test whether deployed security protocols like Transport Layer Security (TLS) are configured correctly (see Section 5.4).

Lastly, our approach of continuous testing follows goals similar to conformance testing, that is, to validate that a cloud service adheres to some controls necessary to obtain some

certificate. While these certificates are usually based on some standards, a construct such a conformance clause is usually missing. In that regard, our framework to design continuous tests can be considered one building block to establish a conformity assessment program.

2.3 Certification

Certification refers to the process where an accredited third party validates that a product, process or service fulfills a finite, predefined set of particular requirements [91]. These requirements are usually referred to as *controls* which are understood as descriptions of measures aiming to modify risk [21] (sometimes certificates' requirements are termed *criteria* [39]) and are defined by a *certification scheme*. If a system complies with all controls defined by certification scheme, then a report called *certificate* is produced, stating *compliance* of the system with the scheme's controls [92].

Certification is a special case of *assurance*, i.e., the process of establishing justifiable confidence that a system behaves as expected and satisfies a set of non-functional properties, e.g., security properties [93]. This *confidence* is derived from *evidence*, that is, observable information of the system which can be collected and evaluated [94]. Although assurance is oftentimes used synonymously with *security assurance*, assurance also includes other properties of a system like reliability and robustness [57].

The motivation for a software supplier to obtain a certification can be driven by the expectation to attract new customers as well as to increase customer retention. Furthermore, as Damiani et al. [59] point out, software customers may not be willing to fully exclude liability when purchasing software. In this context, software suppliers may choose certification as a means to reduce liability in case of, e.g., security breaches resulting from defects of the supplied software.

Numerous national and international standards, guidelines and frameworks exist according to which a product, process or service can be certified, e.g., ISO/IEC 27001:2013 [24], NIST SP 800-53 [95], ISO 9001:2015 [96] or Capability Maturity Model Integration (CMMI) [97]. Organizations like ISO or NIST are considered *certification scheme owners* which usually do *not* conduct evaluations of, e.g., a software product themselves[24]. The task of evaluations leading to issuance of a specific certificate are left to *accredited certification bodies*, that is, specialized organizations which are authorized to conduct assessments and issue as well as revoke certificates [91].

Model-based versus instance-based certification *Model-based certificates* are the result of formal proofs which show that an abstract model, e.g., finite state machines, of a software satisfies a particular property. This approach necessitates that an abstract model is provided supplementary to the actual software or has to be extracted from the source code [59]. *Instance-based certification* relies on *evidence*, i.e., information which can be observed from the software under certification, e.g., through testing or monitoring. This evidence can then be used to make a statement about whether the software under certification possesses a given property or not. Note that such evidence-based statements may not be binary, i.e., a given properties holds or not, but contain a level of uncertainty.

[24] According to Damiani et al. [59], there are some exceptions to that statement, e.g., The German Federal Office for Information Security (BSI) acts as scheme owner and as the certification body.

Damiani et al. [59] point out that the distinction between model-based and instance-based certification is arguable since some researchers in the field of *model-driven testing* such as Ammann and Offutt [61] argue that, e.g., deriving black box test designs – which can be used for test-based certification, one example of instance-based certification – may also make use of abstract models.

Certificate hierarchy A software under certification may be decomposed into subsystems, each of which consisting of components where the certificate issued for that system is hierarchically dependent on the certificates of each subsystem and each component [59]. Such a hierarchy has high practical value in case the system under certification changes since it permits *incremental re-certification*, that is, reusing existing evidence while re-certifying those parts of the system that have changed [28].

2.3.1 Software certification

The notion of software certification was already explored by Mills et al. [98] in 1987 where they propose the *Cleanroom process* which uses testing to demonstrate that an application fulfills some reliability requirements. Further, software certification can be distinguished into three complementing approaches (also known as the *software quality certification triangle*) [99]:

- *Product-based certification:* This approach assesses the software product directly through static and dynamic testing methods [40]. This certification is usually conducted by an independent third party [100] but there are proposals which argue that developers themselves can conduct a form of self-assessment [101]. One of the most well-known certification schemes in context of security certification is the *Common Criteria for Information Technology Security Evaluation* (CC) [102]. Through defining a process and an evaluation framework, CC aims to provide a standard approach to specify, design and evaluate the security properties of a target of Evaluation (TOE), that is, the product or system to be certified [103].

- *Process-based certification:* This approach is also known as *process maturity assessment* which relies on the assumption that a software development process having a set of properties is likely to produce adequate software [104]. A popular examples for process-based certification approach is Capability Maturity Model Integration (CMMI) [97] which was developed by the Software Engineering Institute (SEI) which resides at Carnegie Mellon University (CMU). CMMI succeeds the capability maturity model (CMM) which guides process improvements of organizations through best practice models. Note that conforming with CMMI requirements is referred to as *appraisal* instead of certification. Another example for process-based certification is the Software Process Improvement and Capability dEtermination (SPICE) approach published in form of the ISO/IEC 33001:2015 [105] (formerly known as ISO/IEC 15504).

- *Certification of personnel:* This approach aims at assessing the skills of personnel which is involved in the development of the software. Such assessments can be dependent on the programming language or platform a developer uses, e.g., Java Programmer Certification[25] or be bound to non-functional skills such as security, e.g.,

[25] `https://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=654&get_params=p_id:357&p_org_id=34&lang=D#tabs-2-1` [Accessed: 2018-12-13]

Certified Information Systems Security Professional (CISSP)[26].

2.3.2 Service certification

Erl [106] defines *services* as

> *"[...] individual units of logic to exist autonomously yet not isolated from each other. Units of logic are still required to conform to a set of principles that allow them to evolve independently, while still maintaining a sufficient amount of commonality and standardization.".*

What distinguishes service-based system or service-oriented architectures (SOA) from traditional distributed systems – which may also support message communication and separating interface from process logic – is adherence to the following design principles [106]:

- *Loose coupling:* While dependency between services is kept to a minimum, services need to be aware of other services.
- *Service contract:* Services abide by the communication agreement which is specified by their service descriptions.
- *Autonomy:* Services control the logic they encapsulate.
- *Abstraction:* Aside from what is part of the service description, all remaining logic is hidden by the service.
- *Reusability:* Logic provided by a service explicitly seeks to be reused in different contexts.
- *Composability:* Multiple services can be combined and used as a collection of services.
- *Statelessness:* Services aim to keep required information specific to invocations over time to a minimum.
- *Discoverability:* Services for a desired purpose are to be found with ease.

Due to these principles, it is in some cases – such as large service networks – not trivial to even determine which software components form a service-based system [107].

Approaches to software certification are not suited to support a service scenario [108] because they assume that the system under certification has a stable structure and is running within a stable environment [20]. Further, the methods underlying software certification are designed to evaluate static and monolithic software at installation time where a certificate's representation has to only support human-readable formats [109].

The most important implementation platform for service-based systems are web services [106]. Multiple standards for web services have been proposed and developed, e.g., WS-Security [110] or WS-Reliability [111]; however, these standards describe security requirements which developers should consider during implementation of a service [112], leaving certification of (web) services subject to ongoing research [109]. One example of

[26] `https://www.isc2.org/cissp/default.aspx` [Accessed: 2018-12-13]

these research efforts is the EU-funded Assert4SOA[27] project which developed a framework to support security certification of services through providing machine-readable certificates demonstrating which security properties a web service satisfies [113].

2.3.3 Cloud service certification

Cloud service certification aims to demonstrate that a cloud service complies with a set of controls defined by a certification scheme [20]. Cloud services share some of the service characteristics described in the previous section, e.g., autonomy, abstraction, and reusability. However, when comparing the definition of cloud services presented in Section 2.1.1 with Erl's [106] understanding of services, it is becomes clear that cloud services are rather a wider, more abstract notion. This means that, on the one hand, some cloud service, e.g., a PaaS may make heavy use of web services, thereby inheriting all characteristics of SOA. On the other hand, a virtual machine used remotely via SSH to deploy some application stack – which is considered a typical use case for IaaS – does not fit Erl's definition well. As a result, while cloud service certification may inherit challenges from (web) service certification, it also poses additional, novel challenges due to also having to consider the deployment infrastructure and platform services involved in delivery of cloud services [28]. What holds true for cloud service certification as well, however, is that traditional software certification approaches (see Section 2.3.1) are incapable of taking non-stationary properties of cloud service into account [20].

Standardization activities In 2012, the European Commission in cooperation with the European Union Agency for Network and Information Security (ENISA) published the *European Cloud Strategy* [11] which includes cloud service certification as one integral part to foster adoption of cloud computing within the European ICT sector. In the course of implementing this strategy, the *Cloud Certification Schemes List (CCSL)*[28] has been compiled which consists of certification schemes potentially relevant to cloud customers. Among others, the CCSL includes cloud-specific certification schemes such as the Security, Trust & Assurance Registry (STAR) provided by the Cloud Security Alliance (CSA) [23], EuroCloud StarAudit [114] as well as general standards not specific to cloud services like ISO/IEC 27001:2013 [24]. Not included by the CCSL but nevertheless an important, cloud-specific standard was published in 2016 by the German Federal Office for Information Security (BSI): It is referred to the *Cloud Computing Compliance Controls Catalogue (C5)* [31] and primarily aims at assessing security properties of a cloud service provider.

There are numerous standards, guidelines and frameworks available from which controls to define a cloud certification scheme can be derived, e.g., the Federal Risk and Authorization Management Program (FedRAMP) [30], ENISA's Cloud Computing Information Assurance Framework (IAF) [115] or SP 800-53 [95] published by the National Institute of Standards and Technology (NIST). An overview of such catalogs of controls from the European and German point of view is provided by the Federal Ministry of Economics and Technology of Germany (BMWi)[29] in [116]: According to this study, the NIST was the first standardization body to draft a road map [117] for the standardization of cloud computing.

[27] `http://www.assert4soa.eu/` [Accessed: 2018-12-13]

[28] `https://resilience.enisa.europa.eu/cloud-computing-certification` [Accessed: 2018-12-13]

[29] `https://www.bmwi.de/Navigation/EN/` [Accessed: 2018-12-13]

Continuous cloud service certification While development of standards for cloud services and certification schemes is well under way, conducting certification of cloud services is subject to ongoing research. The need for cloud service certification has been identified by numerous scientific publications, e.g., Khan and Malluhi [118], Ko et al. [119], Sunyaev and Schneider [19] and Cimato et al. [20]. Yet when applying traditional certification to cloud services, the following discrepancy surfaces: 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 [26]. Put differently: Traditional certification assumes that during the period where a certificate is valid, any other evaluation of the cloud service will produce identical results [120]. However, a cloud service may change over time where the changes are hard to predict or detect by a cloud service customer [27]. These changes may lead to the cloud service not fulfilling one or more certificate's controls, 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 capable of *continuously* detecting ongoing changes of a cloud service during operation and assessing their impact on satisfaction of certificates' controls [28][29].

Naturally, *continuously* checking whether a cloud service conforms with a set of controls is not to be understood in a strict mathematical sense: No matter how sophisticated the method to produce evidence, producing evidence will always be – in a strict mathematical sense – carried out by discrete tasks that occur at some point in time. We use the term *continuous* certification to describe *automated* and *repeated* evidence production conducted by a third party which occurs multiple orders of magnitude more frequent compared to traditional certification (e.g., checking property satisfaction every minutes instead of every year).

A special type of continuous cloud service certification is *incremental* certification which aims at reducing costs of re-certification in case of changes of the cloud service under certification [121]. Incremental certification thus adopts the notion of the *Assurance continuity paradigm* of the Common Criteria [122]. An important, implicit assumption that the approach of incremental certification makes is that an event is available which triggers re-certification by indicating that a change of the cloud service under certification occurred as well as the scope of the change.

Research activities Important contributions to advance automated means supporting cloud service certification have been made by the EU-funded research project *CUMULUS*[30] which stands for *Certification Infrastructure for Multi-Layer Cloud Services*. CUMULUS aims to provide a framework to supports certification of IaaS, PaaS, as well as SaaS layer [123]. Another example is the *Next Generation Certification (NGCert)*[31] project, a research project funded by the Federal Ministry of Education and Research of Germany (BMBF). NGCert aims to provide a technical, organizational as well as legal framework to validate that a cloud service complies with controls at operation time. Furthermore, the EU-funded innovation action *EU-SEC*[32] which is short for *European Security Certification Framework* aims at increasing the trust in cloud providers, reducing human interaction through the introduction of mechanisms and tools for continuous auditing, and triggering the adoption of cloud

[30] `https://cordis.europa.eu/project/rcn/105141_en.html` [Accessed: 2018-12-13]
[31] `https://ngcert.de/` [Accessed: 2018-12-13]
[32] `https://www.sec-cert.eu/` [Accessed: 2018-12-13]

service certification throughout the EU. Finally, the goal of the *European Cloud Service Data Protection Certification (AUDITOR)*[33] project is to enable certification of cloud services according to the General Data Protection Regulation (GDPR) of the EU. AUDITOR is funded by the Federal Ministry for Economic Affairs and Energy of Germany (BMWi).

Concepts of cloud service certification A central result of the CUMULUS project is a meta-model describing all relevant concepts of cloud service certification introduced in the CUMULUS deliverable *D2.4 Certification models* [121]. We limit our discussion of this meta-model to a selected set of fundamental classes because a detailed description of all concepts is not required to understand the content of the remaining chapters of this thesis. A comprehensive introduction to this meta-model can be found in the deliverable.

Fundamental classes of the CUMULUS meta-model are introduced hereafter:

- *Property:* The property class is part of the module *Property Vocabulary*: It captures the characteristics of a cloud service which is under evaluation. These properties are derived from the controls of a certificate scheme. Note that the CUMULUS project focuses solely on security properties of a cloud service.
- *Target of Certification (TOC):* This class is included in the module called *certification model*. Instances of this class identify the type of cloud service under certification, i.e., any component involved in delivery of the cloud service. Put differently: The TOC unambiguously describes the scope of the cloud service under certification.
- *Certification model:* This class has the same name as the module including it. A certification model describes which *evidence* is needed to check whether a cloud service, that is, the TOC satisfies a given *property*. Further, it specifies how to produce the required *evidence* and how to use the evidence to reason about cloud service properties using *metrics*.
- *Certificate:* The certificate class is part of the *Certificates* module. A certificate includes the set of controls which the TOC has to satisfy to obtain a certificate artifact. In context of the CUMULUS project, the focus solely lies on certification models which can automatically reason about satisfaction a cloud service's properties. Hence a satisfied control is called *assertion*, i.e., a constraint on a property which is supported by evidence, to indicate that satisfaction can be automatically evaluated.
- *Lifecycle:* This class is also contained in the certification model module. Instances of this class define the possibles states a certificate may assume depending on the results of evaluating evidence, e.g., *issued revoked*, *renewed*, or *upgraded*.

Evidence production methods Certification models are one of the fundamental concepts of cloud service certification introduced in the previous paragraph. A certification model can be classified according to the underlying evidence production method. The following four evidence production methods can be distinguished [20][124]:

- *Monitoring-based evidence production:* These methods use monitoring data as evidence which is produced during productive operation of a cloud-service [125]. We can distinguish between two major types of monitoring-based evidence production

[33] `http://www.auditor-cert.de/` [Accessed: 2018-12-13]

methods: The first group consists of methods proposed by current research (e.g., Krotsiani et al. [28], Schiffmann et al. [126]) which are specifically crafted to check whether particular properties of a cloud service are satisfied, e.g., integrity of cloud service components [126] and correctness of non-repudiation protocols used by cloud services [28]. Those methods require to implement additional monitoring services which are not needed for operational monitoring of the cloud service. The second group consists of existing monitoring services and tools which are used to operate the infrastructure of a cloud service, e.g., Nagios[34] or Ganglia[35]. The data produced by these monitoring tools can also be used as evidence to check a cloud service's properties such as availability [125]. Also data produced by tools which aims to detect intrusions such as Snort[36], Bro[37] or OSSEC[38] can serve as evidence [124][125].

- *Test-based evidence production:* Similar to monitoring-based methods, test-based evidence production also collects evidence while a cloud-service is productively operating. Different to monitoring-based methods, however, test-based methods do not passively monitor operations of a cloud service but actively interact with it through tests. Thus test-based methods produce evidence by controlling some input to the cloud service, usually during productive operation, and evaluating the output, e.g., calling a cloud service's RESTful API and comparing responses with expected results [20][127]. The results produced by these tests then serve as evidence.

  We can distinguish two types of test-based evidence production: The first set of methods includes approaches presented by recent research (e.g., Anisetti et al. [34], Anisetti et al. [35], Ullah et al. [128]) which introduce specialized tests designed to check whether a cloud service satisfies a particular non-functional property, e.g., proper authorization when accessing a cloud service's configuration files [34]. The second set of test-based methods does not explicitly aim at evaluating a cloud service's non-functional properties but rather to compare to cloud services, e.g., based on their CPU-speed [129] or scalability [130]. However, results produced by these tests can serve as evidence to support validation of non-functional requirements related to, e.g., performance and reliability.

- *TPM-based evidence production:* These methods build on trusted computing to provide proofs that the hardware layer of a cloud service's infrastructure is trustworthy [20]. At the center of these methods is the Trusted Platform Module (TPM), a security specification provided by the Trusted Computing Group (TCG)[39] which is implemented as a chip physically integrated with the motherboard of a platform. Since a TPM is implemented in hardware, it provides hardware-level security guarantees, that is, it is resistant to software attacks. A TPM supports secure cryptographic operations such as key generation, encryption, singing, hashing and it can also be used to securely store small amounts of data, e.g., cryptographic keys [36].

  There are two main scenarios for TPM-based evidence production [123]: In the first scenario, the TPM is used to assure that only authorized code is running on hosts which

[34] `https://www.nagios.org/` [Accessed: 2018-12-13]
[35] `http://ganglia.sourceforge.net/` [Accessed: 2018-12-13]
[36] `https://www.snort.org/` [Accessed: 2018-12-13]
[37] `https://www.bro.org/` [Accessed: 2018-12-13]
[38] `https://ossec.github.io/` [Accessed: 2018-12-13]
[39] `http://www.trustedcomputinggroup.org` [Accessed: 2018-12-13]

form the infrastructure of the cloud service [131]. In the second scenario, the TPM itself is not used to directly produce evidence but rather to support monitoring-based and test-based production methods, thereby increasing trust in monitoring-based and test-based evidence [123]. For example, a TPM can be used to verify that a monitoring agent deployed on a cloud component has not been modified [38].

- *Expert-based evidence production:* The previous classes of evidence production methods focus on automatic production of observable facts which can be used to support validation of a certificate scheme's controls. This, however, neglects the status quo of how the certification process is currently conducted within the industry: Production (and analysis) of evidence is left to human experts, that is, accredited auditors [39]. Naturally, those experts may use tools to support their evidence production, e.g., static code analysis tools to conduct code reviews or leverage vulnerability scanners to conduct penetration tests [124].

  Other expert-based methods use specifically crafted questionnaires to permit knowledgeable personnel of the cloud service provider to self-assess their service. A popular example for this kind of evidence production is the *CSA STAR Self-Assessment*: Cloud providers either opt to complete the *Consensus Assessments Initiative Questionnaire (CAIQ)* [132] – which consists of 295 questions cloud customers and auditors may have – or self-assess their compliance with the CSA's Cloud Control Matrix (CCM) [22] or both. A similar approach is followed by EuroCloud Europe[40] which offers a questionnaire which allows cloud providers and customer to self-assess their compliance with the controls of the StarAudit certification scheme [133]. Also in this case, facts obtained from monitoring and test tools can be at least helpful – if not required – to the expert completing these questionnaires.

Note that certification models and their underlying evidence production methods do not exclude each other. On the contrary, different methods can be combined to produce complementary evidence needed to check whether a cloud service satisfies a specific property or not. Such certification models employing multiple, different evidence production methods are referred to as *hybrid* certification models [134].

2.3.4 Certification in context of this thesis

At the center of this thesis are methods which aim to support continuous test-based cloud service certification. We show how to design and represent tests supporting continuous certification (Chapter 4 and 6). Further, we demonstrate the application of our approach within five example test scenarios (Chapter 5) where we support validation of controls derived from BSI C5, CSA STAR, and ISO/IEC 27001:2013, the latter two of which are included in the *Cloud Certification Schemes List (CCSL)*, one result of the *European Cloud Strategy* of the European Commission. Moreover, we also investigate the accuracy and precision of test results produced by continuous tests (see Chapter 7) and investigate how to adapt tests when faced with an adversarial cloud service provider who only pretends to comply with a set of controls (Chapter 8).

[40] `https://www.eurocloud.org/` [Accessed: 2018-12-13]

Chapter 3

Related Work

This chapter describes research efforts which are related to this thesis, identifies gaps and, on this basis, points out the contributions of this thesis to the current status. Recall that in Section 2.3.3 of the previous chapter, we distinguished between *monitoring-based*, *test-based*, *TPM-based* as well as *expert-based* cloud certification models. These models define which and how to produce evidence used to check whether a cloud service possesses a particular property and thus support validation of one or more controls; these models are driven by the underlying evidence production methods. At the center of this thesis is a framework to support *continuous* test-based cloud service certification where the underlying approach to evidence production is special type of test-based evidence production. Furthermore, our framework is designed to support deployment of continuous tests as cloud services themselves.

The next section introduces current approaches to test-based certification while Section 3.2 presents monitoring-based as well as TPM-based certification models. Thereafter, Section 3.3 discusses current work in the area of *Testing as a Service (TaaS)*. Finally, in Section 3.4, we summarize the status quo of related work, point out the gaps of the status quo and explain how this thesis contributes to supplement missing parts.

3.1 Test-based cloud service certification models

Test-based certification models use test-based evidence production methods which produce evidence by actively interacting with a cloud service during its productive operation and evaluating whether the observed behavior conforms with the expected one, e.g., issue a query to a cloud service's database back end and check whether response times do not exceed a defined maximum [20][127].

The next section introduces frameworks which have been proposed by current research to support test-based certification models. These frameworks guide the design of test-based evidence production methods. Then, Section 3.1.2 provides an overview of approaches that present specialized evidence production methods which aim at producing evidence to check particular properties of a cloud service, e.g., security, performance and reliability properties. This section also includes specialized evidence production methods designed to be used when faced with an adversarial cloud service provider. Lastly, in section 3.1.3, we describe approaches which can also produce evidence but whose primary goal is to compare performance of cloud services.

3.1.1 Frameworks to support test-based certification

Anisetti et al. [34] introduce a framework to support testing-based certification of cloud services. They describe testing of five properties, i.e., confidentiality of authentication data, storage confidentiality, network isolation, storage performance, and service performance which are derived from the security guide of OpenStack, developed by the OpenStack Security Group (OSSG)[41]. This work is supplemented and generalized in [35] where Anisetti et al. describe a test-based security certification framework for cloud services. They outline requirements such a framework should satisfy and propose generic components to support implementation of a cloud certification process. They further claim that their framework can be integrated with any cloud stack, address certification on any layer of the cloud stack and may accommodate monitoring-based evidence production techniques. This approach is, again, extended by Anisetti et al. in [135]: In this work, they argue that steps of the certification process are not bound to a single, online certification authority which is always available during the certification process but are conducted by different parties. In order to establish a chain of trust where each party participating in the certification process holds some responsibility, they add a formal method to delegate responsibilities and check the correct execution of steps taken during the certification process at any time. Anisetti et al. [136] apply a similar approach as in [135], however, in this work focusing on autonomic cloud systems, that is, systems which constantly monitor themselves and thus are able to repair and optimize themselves as well as adjusting their non-functional properties [137]. Such autonomic cloud systems are thus able to adapt to changes of their environment over time. It is claimed that in such a setting having only a single certification authority responsible to check correctness of the certification process is insufficient. Therefore, they propose that the certification authority is only responsible for the initial configuration of the certification process and delegates proving correctness to participants of the certification process.

In a different line of work, Anisetti et al. [138] present a model-based approach to re-use existing evidence from existing certificates to limit the amount of new and additional certification with evolving, i.e., changing services. In [139], Anisetti et al. apply the methodology developed in [138] to cloud services: Due to the evolutionary character of cloud services, their environment and certification processes, requiring certification from scratch may invalidate the advantages arising from automated certification. To address this challenge, they introduce a scheme to reduce the amount of testing activities to be executed on cloud services by re-using evidence that has been produced by previous executions of the certification process. It is also claimed that the proposed scheme works on every level of the cloud stack.

Ullah [128] propose a framework to automate security compliance assessment processes, i.e., processes ensuring that a cloud services meets the required level of security. They list challenges to build an automated security compliance tool (ASCT), consisting of identification and collection of relevant audit information, i.e., evidence. A system architecture is proposed, including a cloud audit API to be provided to clients. They propose four ways to collect audit information, that is, evidence: Via existing APIs of the cloud service, using existing vulnerability assessment tools (VATs), log analysis, as well as manual entries, e.g., entered by system administrators. As an example, checks to support validation of two controls of the ISO 27002 are implemented: Clock synchronization and port vulnerabilities. The first one is implemented through adapting OpenStack, the second one uses existing test tools. In a

[41] `https://wiki.openstack.org/wiki/Security` [Accessed: 2018-12-13]

similar effort, Gonzales et al. [140] introduce *Cloud-Trust*, a model to assess and compare the security of cloud infrastructures according to a defined set of security controls.

3.1.2 Specialized test-based evidence production methods

In this subsection, we present current research introducing specialized test-based methods whose evidence aims to check whether a cloud service satisfies particular properties such as security properties or location properties. Also, we include test-based evidence production methods which can be used if the cloud provider whose services are to be certified is not trusted.

Producing evidence to validate security properties Bleikertz et al. [141] propose an approach to validate the correct configuration of AWS security groups. To that end, they use graphs to represent and analyze firewall configuration enforced by virtual routers in AWS EC2 and check if they comply with a defined security policy. Furthermore, they conduct vulnerability assessment of VMs using OpenVas[42], a popular open source vulnerability scanner. They combine their results using weighted attack graphs to assess the vulnerability of a particular EC2 configuration. Whaiduzzaman and Gani [142] outline a concept how to use automated vulnerability assessment of cloud services. Based on the result, they propose to rank the cloud service, thereby enabling customers to choose the most suitable service. Zech et al. [143] present an approach how to derive tests of security properties of cloud services based on negative security requirements. The latter are used to define misuse cases which then serve as a starting point to define corresponding tests. Zech et al. continue this line of work in [144] where they supplement their test generation methodology by generating needed input data to a test using a *generating fuzzer* which creates malicious input used to test the cloud service. Albelooshi et al. [145] present experiments on data remanence in virtual machines arguing that the underlying method can be used by auditors to validate that controls regarding data sanitization are met, thereby preventing so-called *harvesting* attacks. Their focus lies on the scenario that VMs of different customers share the same physical host and when resources of that physical host are re-allocated, then sensitive data is leaked because of the shared memory is improperly sanitized after a VM is terminated. Supporting their claims, they conduct experiments, one of which shows that when repeatedly restarting an AWS instance, memory traces from VMs of other customers can be recovered.

Producing evidence to validate geolocation There is a group of test-based techniques which aim to validate the geographical location where a cloud service or components of a cloud service are hosted. Skandari et al. [146] conduct delay measurements of around 500 websites and use polynomial regression to train a prediction model. Labels required for this supervised learning approach are obtained from geoIP databases while they test their model using three randomly picked destinations. Jaiswal and Kumar [147] propose an application level scheme which combines download times of files and network delay of IP packets to locate the data center where these files are stored. Fotouhi et al. [148] create a set of public landmarks from which geolocation measurements are performed. They use *constraint-based geolocation (CBG)*, a method that was originally introduced by Gueye et al. [149] to locate Internet hosts, which is deployed on EC2 instances to estimate the location of physical hosts within Google data centers. Gondree and Peterson [150][151]

[42]`http://www.openvas.org/` [Accessed: 2018-12-13]

introduce a generic framework to actively validate geographic locations of data in the cloud, using latency-based techniques. To that end, they propose *constraint-based data geolocation (CBDG)* which combines latency-based geolocation techniques with a probabilistic proof of data possession. Fu et al. [152] extend this approach by improving the scheme presented in [151] through using Trusted Platform Modules (TPM). Albeshri et al. [153] present a similar approach called *GeoProof*, a protocol that combines proof of storage protocols with distance-bounding protocols. Proof of storage protocols allow verifying integrity of stored data without completely downloading it while distance-bounding protocols are authentication protocols between a verifier and a prover, serving to establish the claimed identity and physical location of the prover. Ries et al. [154] build on virtual coordinate systems (VCS) which are used to predict network latency without having to conduct extensive measurements. Based on three different VCS, i.e., Vivaldi [155], Pharos [156], and Phoenix [157], they show how to determine the geographic location of virtual machines.

Apart from locating cloud resources as well as data stored in the cloud, there is a set of general techniques which were developed to determine the geographic location of Internet hosts (also referred to as *IP geolocation*). One of the early works on this subject is presented by Padmanabhan et al. [158]. They propose three distinct techniques to locate Internet hosts: (1) *Geotrack* which infers the location using DNS entries and names of nearby network nodes, (2) *GeoPing* which uses network delays to estimate the coordinates of an Internet host, and (3) *Geocluster* uses Autonomous System (AS) information, that is, their prefixes to extract geographical clusters (inter-domain routing information which is derived from Border Gateway Protocol (BGP)). Gueye et al. [149] introduce *constraint-based geolocation (CBG)* which uses a special variant of multilateration to infer the area in which an Internet host is located. Katz-Bassett et al. [159] propose *topology-based geolocation (TBG)* which initially makes a conservative estimate of possible Internet host locations using maximum transmission speed of IP packets. This first estimate is then refined by considering latencies between routers which are on the path from the landmark to the Internet host. Eriksson et al. [160] take a different approach by framing IP geolocation as a machine-learning classification problem. They use a Naive Bayes estimation method to classify IP geolocations. Lastly, Arif et al. [161] present *GeoWeight* which extends CBG by also taking into consideration that for a measured latency to an Internet host, some distances are more probable than others. They use this insight to formalize an additional constraint and show that under certain circumstances, their approach outperforms CBG.

Producing evidence for untrusted cloud service providers Huang et al. [162] present a method to detect cloud service providers which cheat on agreed service level agreements (SLA), in particular on CPU speed. They experimentally evaluate their proposal through testing video conversion time needed for a batch of videos and show that they can detect non-compliant underprovisioning of CPU resources. Houlihan et al. [163] propose a similar approach, also attempting to detect a cloud service provider cheating on promised CPU speed. Further, Ye et al. [164] present a method to detect cheating on promised memory size of a VM. Koeppe and Schneider [165] present how to benchmark performance of cloud service when the cloud service provider is not trusted. They discuss four means of a cloud provider manipulating performance benchmarks and propose tamper resistant benchmark based on *proof-of-work* functions, i.e., functions which are hard to compute but whose results' correctness can be efficiently verified. Juels and Opera [166] introduce an approach to verify the integrity of data stored in a cloud using dynamic Proofs of Retrievability (PoR). To that

end, they built on Iris, an authenticated file system allowing migration of existing internal enterprise files to the cloud.

Producing evidence to validate performance and reliability properties Banzai et al. present *D-Cloud* [167], a distributed test environment which allows to test the reliability and availability of distributed systems, e.g., cloud services. At the center of their approach is a virtual machine called *FaultVM* which is capable of emulating hardware failures. Depending on the service under test, D-Cloud can scale the FaultVM as needed. Pham et la. [168] propose *CloudVal*, an approach which uses fault injection to test the reliability of virtualized environments. They extend the NFTAPE fault injection framework [169] and support fault models such as delay of I/O operations, maintenance events etc. It is shown how path-based and stress-based fault injection can be used to automatically generate and execute fault injections conducted in a black box manner, i.e., without having any knowledge of the implementation of the cloud service under test. Le et al. [170] follow a related approach by injecting faults to the XEN hypervisor and its guest systems. Lu et al. [171] present an approach to remove outliers from noisy data sets which are used to evaluate quality of service (QoS) properties such as throughput, and execution time. They consider two different types of QoS measurements: One is conducted by an objective third party and the other one is obtained by the cloud service provider and cloud service customer. Based on these two types, a global decision model is derived which outputs the optimal service composition given the constraints provided by the two types of measurements. Jayasinghe et al. [172] present *Expertus*, a code generation framework to support automated testing of performance and scalability of large scale distributed applications deployed on IaaS clouds. Building on their previous work on code generation [173], they aim at reducing human error and supporting efficient as well as exhaustive testing of cloud-based applications. Sobel et al. [174] propose a web application which aims to act as a benchmark as well as corresponding load generators and instrumentation. The goal of this approach is to enable test-based evaluation of costs per user per month of web applications when deployed on cloud resources.

3.1.3 Comparative benchmark testing of cloud services

In this section, we present current work whose primary goal is to compare cloud services' performance through benchmarks. Results produced by these benchmark tests can be used as evidence to validate if a cloud service satisfies defined performance requirements.

Li et al. [129] introduce an approach to systematically compare performance of public cloud providers. They point out four major challenges: (1) What to measure? Answer: Performance a customer perceives of a common set of services offered by representative cloud service providers. (2) How to measure a customer's perceived performance? Answer: Select a few important metrics per service, e.g., CPU speed. (3) What use cases to look at such that benchmarking is not too far from practical appliances? Answer: Consider storage intensive e-commerce web service, computation-intensive scientific computing application, and latency-sensitive website serving static objects. Performance measurements of AWS, Microsoft Azure, Google App Engine, as well as RackSpace are conducted. During their discussion of open issues, they also include the notion that snapshots, i.e., measuring and comparing performance based on a single experiment is insufficient and continuous measurement is preferable.

Binnig et al. [175] discuss the challenges and needs of developing benchmarking methods

suitable to evaluate and compare cloud services. They emphasize the need for new metrics reflecting that cloud services may adapt to load changes over time, as well as the need to consider all layers of the cloud service during the benchmark.

Cooper et al. [130] present the *Yahoo! Cloud Serving Benchmark (YCSB)*, a framework which aims to compare the performance and scalability of cloud-based data bases such as Microsoft Azure SQL Database[43] and AWS SimpleDB[44].

3.2 Other cloud service certification models

In this section, we present current research on *monitoring-based* and *TPM-based* certification models. Note that we omit a discussion of *expert-based* certification models, e.g., interviews, questionnaires and so forth because the focus of this thesis lies on automated evidence production methods.

3.2.1 Monitoring-based certification models

This section provides an overview of the status quo of monitoring-based cloud certification models. We distinguish between proposals of frameworks guiding design and implementation of monitoring-based evidence production, specialized monitoring approaches to evaluate particular cloud service properties as well as operational monitoring tools which may also be used as a source for evidence.

Frameworks to support monitoring-based certification Krotsiani et al. [28] propose a framework to support *incremental certification of cloud services*, i.e., a continuous monitoring approach to assess cloud services' security properties using operational evidence, that is, operational data. To that end, they present a lifecycle model for incremental certification: After activation of a certificate type which specifies a certain certification model, a suitable monitoring infrastructure is configured to obtain operational evidence. Then required monitoring events and functions are determined, and monitoring is deployed. Evidence is accumulated until the certificate is satisfied, then the certificate is issued.

Schiffman et al. [126] propose a framework to enable customers to obtain a verifiable chain of trust that tracks runtime status of their cloud instances. The framework has to be deployed by the cloud service provider and consists of two main parts: The first one provides integrity proofs on the platform layer where system images are designed and distributed to the nodes of a cloud. Whenever such a node wants to join the cloud, it has to undergo remote attestation. The second part consists of a service running on every compute node monitoring integrity measures (e.g., load-time and runtime properties) available through interfaces of, e.g., virtual machine introspection (VMI) and virtualized TPMs (vTPM). It is assumed that a customer can verify integrity measurements at runtime through suitable protocols.

Wang et al. [176] present a system to collect and evaluate audit log data within a cloud infrastructure. The system is called *CDCAS* and comprises four main components: Collector, controller, analyser, and dashboard. Log collection itself possesses a logical architecture consisting of a three-layered model: Agent layer which gathers log data, transform layer which gathers data from agent nodes, and server layer which analyses data and makes backups. Controller detects changes in the infrastructure and deploys suitable agent configurations for

[43] `https://azure.microsoft.com/en-us/services/sql-database/` [Accessed: 2018-12-13]

[44] `https://aws.amazon.com/de/simpledb/` [Accessed: 2018-12-13]

them, e.g., if new VM is started. Analysis is supported by mining techniques to generate new signatures. They use frequent patterns in log data from which security relevant events are extracted.

Doelitzscher et al. present *Security Audit as a Service (SAaaS)* [177][178], a framework which uses autonomous agents deployed within cloud environments that automatically validate security status and compliance of IT security policies. Agents are placed at selected points of the cloud infrastructure, e.g., within virtual machines, VM hosts, or network devices. Each agent possesses a set of rules specifying actions in case a trigger event is observed, thus an agent implements policy and enforcement functionality. Status reports derived from events are presented to users as evidence.

Haeberlen [179] demands that cloud services have to be *accountable*, that is, an approach is needed to detect faults and link them to their source, e.g., a faulty node, in a non-disputable manner. To that end, he proposes four building blocks: (1) Tamper-evident logs, (2) virtualization-based replay, i.e record non-deterministic inputs to a VM and replay on different instance, (3) trusted timestamps as well as (4) sampling. In [180], Haeberlen et al. implement their proposition, presenting *accountable virtual machines (AVM)*. Driven by a similar motivation, Ko et al. [181] also propose to use centralized logging of operations of cloud service components as a means to establish accountability of cloud services.

Accorsi et al. [182] propose *ComCert*, a tool which uses petri nets to support auditors to check whether execution of business processes of a cloud service provider complies with a set of controls. In a related line of work, Awad and Weske [183] build on their previous work in [184] and use BPMN-Q queries to detect violations of compliance rules in respect to the sequence of executing activities.

The work of Yao et al. [185] is orthogonal to the above framework proposals: They assume that admitting to a compliance violation by a participating entity, e.g., the provider of a cloud service, may cause penalties, thus creating the incentive for the violating party to deceive. They propose a so-called *accountability service* which acts as a trusted third party keeping all the signed logging data, thereby guaranteeing non-repudiation of logging data. They use BPEL to model business processes and insert a logging invocation after each BPEL activity.

Specialized monitoring-based evidence production methods Continuing their work in [28], Krotsiani and Spanoudakis [33] propose to use continuous monitoring to verify the correctness and robustness of non-repudiation protocols deployed within cloud systems. They present two main monitoring rules which are represented as EC-Assertions. Aside from modeling the non-repudiation property itself, three other monitoring rules are introduced which support the assessment scheme, a part of the monitoring-based certification model. This model defines conditions regarding the evidence, it consists of: Sufficiency of collected evidence, expiration period for certificates, and anomalies as well as conflicts to be monitored during the certification process. Thus, the three complementary monitoring rules aim at detecting potential DoS attacks, suspicious behavior, and anomalous operating conditions, e.g., the network delay which may affect the response time measured in the first place and possibly leading to time outs, i.e., a violation of the non-repudiation property.

Birnbaum et al. [186] propose behavioral modeling on VM and hypervisor level to enable timely assessment of security incidents of cloud services. To that end, they monitor system calls at the hypervisor level and extract a benign profile of an application. These benign profiles are represented using colored petri nets and serve to detect anomalous behavior.

Park et al. [187] provide an overview of cloud auditing techniques suitable for real-time audits of cloud services, outline challenges and needs, and inspect AWS CloudWatch[45] to discuss possible extensions to better support audits. Driven by a similar motivation, Nix et al. [188] propose to use a machine learning approach to support online audits of cloud services. To that end, they outline an incremental data mining approach using Very Fast Decision Tree (VFDT). The idea is that through acquiring a model by data mining techniques, it is possible to save time and space by estimating results of queries instead of actually executing a query on data streams.

Aside from audit specific approaches, several general approaches to monitor cloud services and infrastructures have been proposed, attempting to meet cloud characteristics such as scalability (e.g., Wang et al. [189], Clayman et al. [190], Anand [191], Zareian et al. [192], Kai et al. [193]), finding suitable abstractions from detailed, heterogeneous monitoring data (e.g., Shao et al. [194], Zhao [195]), providing trusted monitoring data (e.g., Sundareswaran et al. [196], Zou et al. [197]), proposing generic cloud monitoring architectures (e.g., Chaves et al. [198]) as well as detecting security incidents (e.g., Monfared and Jaatun [199], Khorshed et al. [200], Krautheim [201]). Comprehensive surveys on cloud monitoring in general are provided by, e.g., Aceto et al. [202] and Mohamaddiah et al. [203].

Most of the above cloud monitoring solutions (e.g., Kai et al. [193], Clayman et al. [190]) explicitly suggest that monitoring data can be used to check whether service level agreements (SLAs) hold. Cloud certification schemes usually contain controls which require that SLA are defined and satisfied, e.g., *CCC-05* of the CSA's Cloud Control Matrix (CCM) [22]. Starting from the angle of a SLA, Mahbub and Spanoudakis [204] present a framework which uses *EC-Assertions*, a first-order temporal logic language based on Event Calculus, to monitor and reason about satisfaction of SLAs. Later, Spanoudakis et al. generalize this approach in [205] by introducing the monitoring framework called *EVEREST*. They apply their approach to cloud services in [206]. In a related line of work, Foster and Spanoudakis also present an approach how monitoring configurations can be derived from SLA specifications [207]. Lastly, Foster and Spanoudakis present *SMaRT* [208], a discovery and configuration tool which checks if and to what extent an infrastructure supports monitoring SLA satisfaction, and, if sufficient, configures the monitoring infrastructure accordingly.

Evidence produced by operational monitoring tools There are various monitoring tools available which are used to monitor large distributed systems such as cloud infrastructures. Alhamazani et al. [209] provide an overview of commercial cloud monitoring tools. Popular tools are Ganglia[46] (developed by Massie et al. [210, 211] and Sacerdoti et al. [212]), Nagios[47] and MonaLisa[48] (proposed by Newman et al. [213] as well as Legrand et al. [214]). Also, tools used to detect intrusions such as Snort [215][49], Bro[50] [216] and OSSEC[51] can be used as a source for evidence [124][125].

Furthermore, cloud services also provide their own proprietary monitoring APIs. AWS, for example, offers CloudWatch[45] to customers and *CloudMonix*[52] (formerly *AzureWatch*)

[45] `https://aws.amazon.com/cloudwatch/?nc1=h_ls` [Accessed: 2018-12-13]
[46] `http://ganglia.sourceforge.net/` [Accessed: 2018-12-13]
[47] `https://www.nagios.org/` [Accessed: 2018-12-13]
[48] `http://monalisa.caltech.edu/` [Accessed: 2018-12-13]
[49] `https://www.snort.org/` [Accessed: 2018-12-13]
[50] `https://www.bro.org/` [Accessed: 2018-12-13]
[51] `https://ossec.github.io/` [Accessed: 2018-12-13]
[52] `http://www.cloudmonix.com/` [Accessed: 2018-12-13]

which aims at monitoring Microsoft Azure cloud services. These monitoring tools focus on providing operational information such as system load, network I/O etc. However, monitoring data produced by these tools can be used to reason about controls related to availability, reliability and performance of cloud services [125].

3.2.2 TPM-based certification models

This section presents current approaches to use Trusted Platform Modules (TPM) to produce evidence. Bertholon et al. [37] introduce *CertiCloud* which aims to protect IaaS using Trusted Computing. Their approach centers around two protocols: The first one is referred to as *TPM-based Certification of a Remote Resource (TCRR)*. This protocol establishes that only authorized code is executed by the remote system. The second protocol is called *VerifyMyVM* permitting a user to check the integrity of the system at runtime to detect any unauthorized modification of the system's configuration. Chuanyi et al. [217] and Xiang et al. [218] pursue a similar approach to use remote attestation to assess the integrity of applications deployed on virtual machines at runtime. Also, Muñoz and Maña [38] follow a similar approach. They propose a method which uses TPMs to combine software with hardware security certification. Different to the above approaches, Khan et al. [219] propose a method for cloud customers to verify the integrity of cloud nodes, i.e., hosts on which virtual machines are deployed, before they start operating on them. Also, Ruan et al. [220] propose the use of remote attestation as a means to establish integrity of cloud resources. Their approach differs from those previously described because they do not rely on a single trusted third party but on *decentralized attestation*: They treat nodes within the cloud as a peer-to-peer network where each node remotely attest each other and share this evidence with all participants, thereby effectively creating a web of trust.

TPM-based evidence to validate geolocation TPM-based evidence production methods are also explored when the goal is to validate the location of a cloud service or cloud service components such as underlying servers. In this context, Noman and Adams [221] outline a scheme to establish a trustworthy binding between location and a physical server by combining GPS data with a TPM. A similar approach is described by Yeluri and Castro-Leon [222], they propose to store a geo tag in any physical server used by a cloud service provider where the tag is supposed to be kept directly in the TPM for later remote attestation. Vaish et al. [223] also propose a TPM which is configured with location data at installation time and later used to compare these coordinates with secure GPS data obtained at runtime.

3.3 Testing as a service

Gao et al. [224] give an introduction to cloud testing, defining cloud testing as

> *"[...] testing and measurement activities on a cloud-based environment and infrastructure by leveraging cloud technologies and solutions."*.

They distinguish the following types of cloud testing: Functional, integration, API and connectivity, performance and scalability, security, interoperability and compatibility, as well as regression testing. They also highlight challenges of cloud testing such as testing security properties of cloud services and integration of testing components with cloud services. Continuing this line of work, Bai et al. [225] investigate existing cloud testing

tools where they distinguish between existing test tools deployed as part of cloud services, academic tools such as *D-Cloud* [167], *CloudSim* [226], PreFail [227][228] and *Cloud9* [229] as well as cloud-specific benchmarks, e.g., *CloudStone* [174], *MalStone* [230] and *YCSB* [130]. Although they identify the need for cloud testing to support SLA validation as well as continuous testing of SaaS applications, Bai et al. do not explicitly consider using test results as evidence to validate controls. Later, Gao et al. [231] make a concrete proposal how to design and implement an infrastructure for TaaS. In this context, they also point out that – as part of future work – they will extend their architecture to support continuous testing of SaaS applications. In a related line of work, Gao et al. [232] review the concept of *testing as a service* by comparing TaaS with conventional software testing as well as discussing requirements and challenges. Tsai et al. [233] build on this work by discussing and comparing different designs of TaaS architectures while Bai et al. [234] present *Vee@Cloud*, a platform-agnostic test environment that leverages cloud resources to enable on-demand as well as scalable testing capabilities. They also point out that existing, continuous monitoring services that are provided by cloud service providers themselves aiming at, e.g., demonstrating conformance with SLAs, fall short on comparability since performance indicators defined by different cloud providers may deviate. They argue that their test environment can alleviate this shortcoming by supporting uniform cloud performance models which can be continuously tested.

Candea et al. [235] also outline the notion of *automated testing as a service* by presenting three use cases they consider major: (1) Support developers to improve their code through more comprehensive as well as continuous testing by leveraging cloud resources, (2) allow users installing a software to validate whether this software meets their requirements and (3) support certification to independently assess the properties reliability, safety and security of software products, thereby allowing to compare software vendors. In a similar effort, Yu et al. [236] introduce a framework to support testing as a service. They extend this line of work in [237] where they introduce a tool which uses cloud resources to automatically execute tests.

Ciortea et al. [229] present *Cloud9*, a cloud-based testing service which employs symbolic execution[53]. They point out that the main challenge lies in building a parallel symbol execution engine because – when naively applied – symbolic execution suffers from extensive memory and CPU usage. They demonstrate how to parallelize symbolic execution, thus being able to use cloud resources to execute the tests faster.

Parveen et al. [239] present a distributed execution framework called *HadoopUnit* which allows running unit test in parallel and thus increase test completion time. In a related line of work, Parveen et al. [240] discuss the question which are the determining factors when considering deploying testing as a cloud service. They consider the specific characteristics of the application under test as well as the types of testing currently used to evaluate the application.

Baride and Dutta [241] outline the concept of using cloud resources to support testing of mobile applications. Also focusing on a specific type of application, Ganon and Zilbershtein [242] present an approach to use IaaS resources to conduct test of Network Management Systems (NMS) which manage a large set of elements, distributed across a network, e.g., VoIP clients.

Lastly, Riungu et al. [243] contribute to research efforts in the field of testing as a service by taking a different angle: They conduct an empirical study by interviewing professionals to

[53]Symbolic execution uses *symbolic* instead of regular inputs to run an application which is more efficient that exhaustive input testing, yet delivers equally complete results [238].

determine conditions that influence software testing as a service as well as to identify future research directions. They found out that domain-specific knowledge required to design tests might hinder adoption of TaaS and, further, that security and safety of the test service is vitally important, i.e., keeping test results secure from unauthorized access and being able to deliver testing as promised. Also, regulatory requirements may prevent companies from exposing their internal testing process to an online testing service.

3.4 Summary and identification of gaps

In this section, we summarize the related work presented in this chapter and point out the gaps. We also outline how the remainder of this thesis will address the identified gaps.

Continuous certification Proposed frameworks to support test-based certification of cloud services ([34][35][135][136][138][139][128][140]) presented in Subsection 3.1.1 provide means to guide design and implementation of automated testing. However, none of those frameworks incorporate the notion which explicitly allows to control repeated and automated execution of self-adaptive tests. Furthermore, although specifically crafted to automatically produce evidence to reason about specific cloud service properties, specialized test-based evidence production methods introduced in Subsection 3.1.2 also do not explicitly integrate means to continuously execute tests. As a consequence, current research lacks a comprehensive discussion as well as approaches on how to reason about sequences of test results produced by continuous tests of cloud services.

It is important to note at this point that monitoring-based and testing-based certification models are complementary means to check whether a cloud service has a set of properties [134]. In the course of the CUMULUS project [121], the term *incremental certification* has been coined which, as already pointed out in Subsection 2.3.3, implies that events are available which trigger re-certification in case a cloud service changes over time. In fact, according to the results of the CUMULUS project, any monitoring-based certification model follows the notion of incremental certification. Thus, all monitoring-based evidence production methods introduced in Subsection 3.2.1 can be considered supportive of continuous monitoring-based certification (e.g., [28][33][126][176][178][180][185][182][183][184][185]).

This thesis will address these gaps as follows: We propose a framework to guide the design of tests to support continuous test-based certification of cloud services (Chapter 4). This framework is orthogonal to specialized test-based evidence production methods, that is, such methods can leverage our framework to be executed continuously. We do not constraint the implementation of our framework by presenting a specific, mandatory architecture. Instead, we make use of formal languages to define a domain-specific language with which continuous tests can be rigorously described and configured (Chapter 6). Furthermore, we introduce universal metrics which can be computed on the basis of sequences of test results produced by any continuous test (Chapter 4). These universal metrics allow us to evaluate and compare the accuracy and precision of continuous test results (Chapter 7).

Invasiveness of evidence production methods Approaches to monitoring-based certification presented in Section 3.2.1 are usually inherently *invasive*, that is, they require to structurally alter the infrastructure of the cloud service under certification (e.g., [33][126][176][178][180]). Even in the case where monitoring-based evidence production methods make use of operational monitoring data ([210, 211][212][213][214][215][216]), this data has to be

made available to the evidence production method in a well-defined manner, e.g., through designated APIs.

Despite test-based evidence production methods having the potential to be deployed in minimally invasive manner (e.g., [162][163][164][147][148][154][129]), current proposals of frameworks to support test-based cloud certification ([34][35][128][140]) require changes to the structure and configuration of the infrastructure components involved in delivery of a cloud service under certification. Furthermore, the idea of *incremental test-based certification* has been outlined in the course of the CUMULUS project [121]. Such an approach can be classified as continuous white box testing (see Subsection 2.2.1) since some event is needed to trigger re-execution of tests. This implies that the cloud service under certification has to be designed or adapted in a way such that required events triggering re-testing can be supplied to the test-based evidence production method. Using continuous black box testing as an evidence production method, however, has not been considered by research efforts so far.

This thesis will address these gaps as follows: Our framework to support continuous test-based certification is designed to support *non-invasive* as well as *minimally invasive* testing, that is, requires minimal or no changes of the infrastructure of cloud services under certification (Chapter 4 and Chapter 5). Yet our framework also supports designing and implementing invasive tests; therefore, our approach can be understood as a generalization of current research on frameworks supporting test-based cloud service certification. Furthermore, the framework allows designing continuous black box tests which build on expected behavior as derived from interpretations of a certificate's controls. Again, having no knowledge about the internal mechanisms of the components of the cloud service under test is a rather strict assumption and our framework also permits some form of continuous white box testing where knowledge about the internal structure is available during test design.

Testing as a service In Subsection 3.3, we presented current research which aims to leverage cloud resources to provide cloud-based testing services. These approaches focus on optimizing test execution time as well as supporting testing on a large scale ([234][229][239][241][242][237]). Although some of these approaches bring forward the notion of continuous testing as a service (e.g., [231][234]) as well as point out using testing as a service to check whether a cloud service complies with reliability and security requirements (e.g., [234][235]), none of the current research effort present solutions to either of those challenges.

This thesis will address these gaps as follows: The design goal of minimally to non-invasive testing already implies that an implementation of our framework to support continuous test-based certification models is not part of the infrastructure of the cloud service under certification. We deploy continuous tests on a remote host using IaaS (Chapter 5), thereby rendering our approach testing as a service by definition. Furthermore, our example continuous test scenarios aim at validating that a cloud service complies with controls related to availability, reliability and security properties.

Untrusted cloud service provider Some specialized evidence production methods do not assume a fully trustworthy cloud service provider and adapt their methods accordingly ([162][163][164][165][166][185]). However, these approaches only focus on isolated adversarial strategies a cloud provider may select. As a result, they do not provide a general approach to model the behavior of an adversarial cloud provider which allows to reason about adaptations of evidence production methods in general.

This thesis will address this gap as follows: We introduce a model to reason about the behavior of a fraudulent provider who only pretends to comply with the controls of a certificate when, in fact, he is not (Chapter 8). Yet, this provider does not cheat arbitrarily, on the contrary: He only cheats if he is sure that he will not get caught. We show how this behavior model allows to derive countermeasures in the form of adaptions of test-based evidence production methods and describe how our framework integrates these countermeasures.

Chapter 4

A framework to support continuous test-based cloud service certification

This chapter presents a framework to design *continuous*, i.e., *automated*, *repeated* and *self-adaptive* tests of cloud service properties, thereby addressing *Research Challenge 1: Design of tests supporting continuous cloud service certification* described in Section 1.2.1. The primary goal of these tests is to continuously produce results which can be used as evidence to support continuous certification, that is, support continuously checking if a cloud services adheres to a set of controls. Parts of this chapter are published in [127] and [244].

It is important to note that the proposed framework in this chapter focuses on providing the technical means to design continuous tests. However, it does not specify a process how to manually or automatically conceive feasible test designs for a concrete cloud service instance. Also, as already pointed out in Section 1.2.1, it is outside the scope of this work to bridge the semantic gap between controls and tests, i.e., provide a method to rigorously derive test from control descriptions.

In the following section, we describe our overall approach which we follow to develop the framework. On this basis, Section 4.2 defines the requirements which guide the design of the framework. Thereafter, Section 4.3 introduces the framework's building blocks and Section 4.4 outlines one example implementation of the framework which is called *Clouditor*. Lastly, Section 4.5 summarizes as well as discusses the contents of this chapter.

4.1 Approach

In order to develop a framework supporting continuous test-based cloud certification, we define the main objective such a framework pursues as follows:

> *Support design of tests which permit to continuously evaluate properties of existing cloud services where test results serve as evidence to validate that the service satisfies a set of controls.*

Based on this main objective, the following section derives requirements the framework has to fulfill in order to achieve the following subgoals:

- Production of evidence,
- extensibility,

- flexible integration with existing infrastructures of cloud services,
- independence of cloud service model,
- reusability of continuous test components,
- integration of existing test tools, and
- self-adaptivity.

Based on the elicited requirements, Section 4.3 defines the building blocks of the framework. For each building block, we describe how it addresses the requirements defined in Section 4.2.

4.2 Requirements

This section describes the requirements which guide the development of our framework's building blocks to design continuous tests.

4.2.1 Production of evidence

Producing results which can serve as evidence to check whether a certificate's control is satisfied is the most important requirement which our framework has to fulfill. Recall that evidence is produced when cloud service properties are evaluated, in our case through using continuous, i.e., automated and repeated testing. Without any further analysis, the test results themselves may only possess insufficient semantics, thus not allowing to reason about controls' satisfaction. Consider, as an example, that a continuous test which aims at testing the availability of a cloud service produced three successive test results indicating that the service was not available. These result cannot directly be used to check a control which demands satisfaction of service level agreements which, e.g., contain the statement that the service is available for at least 99.9999% per month. Therefore, our framework needs to provide means to analyze test results observed over time such that they can be used to check whether controls are met.

4.2.2 Extensibility

The ability to add novel tests to validate new or changed controls or to provide alternative tests to validate existing controls is at the core of the framework. Further, cloud services under test may change to an extent which renders deployed tests unsuitable, thus requiring to design and execute alternative tests.

There may be a common set of tests which can be used with any cloud service to evaluate a basic set of controls any cloud services should satisfy. However, a cloud service provider can be a service customer at the same time creating cloud-based applications using another cloud service provider's resources. As a result, the framework needs to support adding novel tests which are specific to a particular, individual cloud-based application.

4.2.3 Flexible integration with existing infrastructures of cloud services

The design of continuous tests is primarily driven by the controls whose satisfaction they aim to check. However, a cloud service provider's willingness to support a particular tests may vary depending on which and how many changes to the cloud service's infrastructure are needed. On the one hand, in a rather restrictive scenario, a test may only have the same access privileges to a cloud service as are provided to regular service customers. On the other hand, in a less restrictive scenario, the test may be deployed as part of the cloud service's infrastructure and granted the same access level privileges as, e.g., operational personnel maintaining the service. A framework to design and execute continuous tests of cloud services has therefore to be able to support the following levels of integration:

- *Non-invasive integration:* As the name indicates, this type of integration requires no change of the productive environment which is used to operate the cloud service under test. This means that a test can produce evidence without requiring any changes to the cloud service. This type of integration implies that the test's implementation does not become part of the cloud service infrastructure but operates on a remote host, external to the cloud service's infrastructure.

  Consider, as a basic scenario, the endpoint of a SaaS application, i.e., a web site which is publicly reachable. In order to produce evidence to check if this endpoint only supports secure communication via HTTPS with its users, no changes to the SaaS application are needed. As a different example, consider a SaaS application to which only authorized user have access. In order to test that, for example, any input fields available to authorized users properly validate user input and thus do not possess some SQL injection vulnerability, user level access privileges are required. However, none of these tests require to change the composition or configuration of the production environment of the cloud service.

- *Minimally invasive integration:* This type of integration requires to change the configuration of the production environment of the cloud service under test to permit production of evidence. Similar to non-invasive integration, minimally invasive tests are not deployed as part of the infrastructure of the cloud service under test.

  As an example, consider changing security groups to allow a remote host sending TCP segments to a cloud service component, e.g., a virtual machine in order to check its responsiveness. The genuine security model of the cloud service may not allow some components to be accessed from external hosts which are not part of the cloud service's infrastructure. Therefore, the configuration of the security groups have be altered so that TCP traffic originating from the remote host where the tests are executed can reach the service components under test. Another example of minimally invasive integration consists of allowing tests to call APIs of the cloud service to check, e.g., if an encryption policy is defined for objects persisted in the object storage of the cloud service. Such configuration checks on the control plane of the service require adding a designated role granting the test access rights to issue the desired API calls.

- *Invasive integration:* This type of integration requires to change the composition of or the applications used by a cloud service's productive environment to allow tests to continuously produce evidence. Contrary to non-invasive and minimally invasive integration, invasive integration implies that at least part of test's implementation is

deployed as part of the production environment which is used to operate the cloud service under test. The following subtypes of invasive integration can be distinguished:

- *Compositional changes:* In this case, structural changes to the cloud service composition are needed such as adding a virtual machine or micro service where the evidence production technique is deployed and operating on. A classic example of invasive integration through compositional changes are monitoring agents, that is, additional applications which are deployed on virtual or physical components of the cloud service.
- *Code-level changes:* Here, changes in the form of patches to applications which constitute components of cloud services are needed in order to produce evidence. As an example, consider changes of the scheduler of a cloud platform management system such as OpenStack to be able test if deployments of virtual machines which ought to be started only on designated hosts indeed do not share the underlying hardware with other machines.

4.2.4 Independence of cloud service model

As described in Section 2.1.1, cloud services can be of type IaaS, PaaS, SaaS or combinations thereof. Our framework needs to be able to support test-based certification of components of any of these types. More specifically, supporting test-based certification according to a set of controls translates to continuously evaluating properties of cloud services using tests. Therefore, our framework needs to be capable of testing properties of components of any cloud service model.

4.2.5 Reusability of continuous test components

Checking satisfaction of similar controls may lead to similar designs of tests. Thus, our framework has to support reusing parts of an existing test to allow for designing a new, similar test. Furthermore, validation of a control may require combining parts of existing tests. Also in this case, our framework has to have a modular structure which permits singling out and recombining parts of tests.

4.2.6 Integration of existing test tools

Numerous tools exist which can be used to test components and interfaces of cloud services. As an example, consider leveraging a penetration testing tool such as *SQLMap*[54] to detect SQL injection vulnerabilities as part of a continuous test. The results of such a test can support certification according to security controls which demand that the web application components of a SaaS application have to implement state-of-the-art user input validation mechanisms.

The key insight here is that domain experts, e.g., security experts, have built test tools and frameworks that we can take advantage of. In order to utilize the expertise used to build these test tools, our framework has to support integration of such tools when designing tests. Naturally, these tools are not designed with the focus of detecting whether a cloud service

[54] `http://sqlmap.org/` [Accessed: 2018-12-13]

satisfies some certificate's controls. This leads us back to the central requirement described in Section 4.2.1, that is, production of evidence.

4.2.7 Self-adaptivity

A continuous test has to be self-adaptive in two ways: First, it needs to be able to adapt the frequency of repeatedly executed tests within a particular period of time. This allows us, e.g., to accurately evaluate controls with temporal constraints: Once a test has produced evidence indicating that a control is not satisfied at a time, we increase the frequency of following, repeatedly executed tests. That way, we achieve higher accuracy when estimating the period during which a control was not satisfied. Consider, as an example, a test checks every minute whether a cloud service is reachable via HTTP. At a some point, this test fails, indicating that the service is not reachable. The following tests are executed every ten seconds in order to more accurately measure the period of time during which the service was not reachable. Once a test passes again, the next tests are executed every minute again.

Second, assumptions about the environment of a continuous test can be stationary. Thus, these assumptions are rendered incorrect if the environment conditions change over time. If ignored, a test may produce inaccurate results, that is, it may incorrectly fail because of changes in the test environment. Thus, a continuous test needs to be able to detect and adapt to changing environment conditions to produce accurate test results. Recall the above example of testing a cloud service's reachability via HTTP. Let's assume that a test failed because a network failure for which the service provider is not responsible. In this case, we cannot use the results produced by the test to reason about the cloud service's property. Consequently, the test has to detect such changes of the test environment and, e.g., discards the test results.

4.3 Framework building blocks

This section introduces the five main building blocks of our framework to support continuous test-based cloud certification. We begin with an overview of the core concepts and lay out how they can be used to design a continuous test (Section 4.3.1). Thereafter, we explain each building block in detail (Section 4.3.2 – 4.3.6).

It is important to note that the notation introduced in this chapter to define the building blocks serves as a convenience to concisely describe each block. Furthermore, we make use of this notation to describe the test designs underlying each of the example continuous test scenarios presented in Chapter 5. Lastly, we draw on this notation in Chapter 6 to derive a domain-specific language to rigorously define continuous tests.

4.3.1 Overview

Our framework automatically and repeatedly executes self-adaptive tests to support continuous evaluation of statements about cloud service properties. A continuous test CT consists of five building blocks: *Test suites* ($\mathcal{TS}$) define any single test which is executed repeatedly within a continuous test. When defining a test suite, one or more test cases ($\mathcal{TC}$) are associated with the suite. A *test case* forms the primitive of any 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* ($\mathcal{M}$) which allows us to evaluate statements over a cloud service's property, e.g., a detected security vulnerability of a cloud service is fixed within 24 hours. On the other hand, test suite results are used by the *workflow* ($\mathcal{W}$) 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* ($\mathcal{TP}$).

Figure 4.1 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.

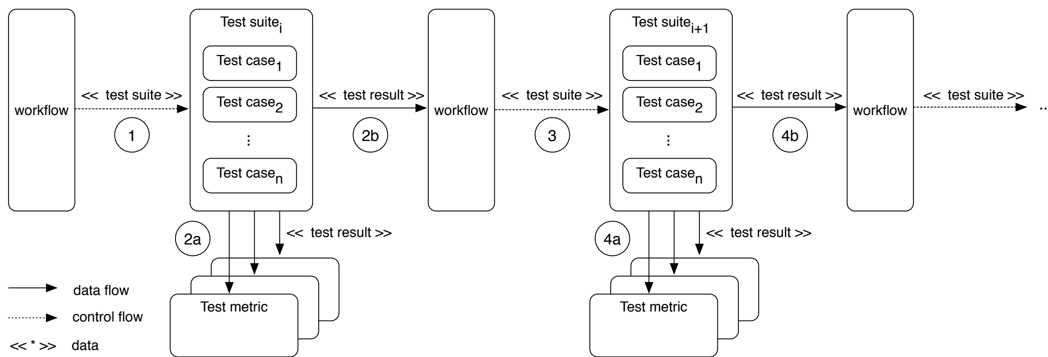


Figure 4.1: Overview of building blocks to support continuous test-based certification of cloud services

4.3.2 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 the path to the 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 4.3.3). Consider, for example, a test suite which has three test case 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 (4.1) 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 $N \in \mathbb{N}^+$:

$$TC = \langle E, L, O, N \rangle. \tag{4.1}$$

Recall the test case example of connecting to a VM and installing a package: This test case may contain the three procedures $E = \langle \textit{connect_via_ssh, install_package, compute_mac} \rangle$ where a `SSH` connection requires input parameters $P_1 = \langle \textit{username, hostname, path_to_private_keyfile} \rangle$, 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: $O = \langle \textit{compare_mac} \rangle$. 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 trusted package installation (TPI) test case as follows:

$$\begin{aligned} TC^{TPI} = \langle & \langle connect_via_ssh, install_package, compute_mac \rangle, \\ & \langle \langle username, hostname, path_to_private_keyfile \rangle, \langle package_name \rangle, \langle key \rangle \rangle, \\ & \langle compare_mac \rangle, \\ & 1 \rangle. \end{aligned}$$

As mentioned above, arguments passed to an input parameter can be randomized. As an example, consider the input parameters $p_{21} = \textit{package_name}$ and $p_{31} = \textit{key}$ 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 our framework requires executions of test cases to be independent of each other, that is, whether a test case is executed or not does not depend on other test cases' results. However, we note that concurrently executing multiple test cases on the same service naturally can produce side effects, i.e., test case results that affect each other.

Corresponding requirements Test cases are the central building block of the framework which enables designing novel tests (*extensibility*). Each procedure of a test case can be implemented using external tools, e.g., using an existing SSH library to connect to a virtual machine and test whether the session has been established successfully. Thus, test cases can be used to *integrate with existing tools*, that is, execute these tools and use the returned results to reason about cloud services' properties. In this context, test cases can be designed

to evaluate properties of SaaS, PaaS, or IaaS, or combinations thereof, thus rendering our framework *independent of cloud service models*. Lastly, execution of test cases can be tightly integrated with existing cloud service infrastructure, entailing privileged access to, e.g., proprietary API of the cloud services. Alternatively, test cases can be designed in a way that is minimally invasive, only requiring minimal changes to the existing cloud service and having non-privileged access. Therefore, our framework allows tests to be *flexibly integrated with existing infrastructures of cloud services*.

4.3.3 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 the 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, the workflow can trigger successive executions of a test suite infinitely often *unless* the workflow decides – based on previously observed test suite results – to execute a different test suite next (details of the building block *workflow* are described in Section 4.3.4).

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 be defined as a range from which the start of a test suite's execution is selected randomly. For example, the interval $\langle 0, 60\rangle$ defines that execution of the next test suite will be randomly triggered within a time interval of 60 seconds after the previous one completed. Further, an interval can be specified as a sequence of values where each value maps to the current iteration of the test suite, thus rendering the waiting time between successive test suite executions iteration dependent. Consider, for example, the interval $\langle 5, 10, 30\rangle$ which defines to wait five seconds until execution of the test suite if it is executed for the first time, ten seconds if it is executed for the second time and so forth. This implies that the number of specified iterations for a test suite cannot be greater than the number of elements of the sequence defined for the interval.

Lastly, if subsequent iterations of a test suite start instantaneously, then they may produce unwanted side effects. In order to prevent such side-effects, 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 (4.2) 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 $I \in \mathbb{N}^+$, and the offset $F \in \mathbb{N}^+$ (seconds) between test suite execution. Further, a test suite consists of a tuple of interval elements T where each $t \in T$ specifies the waiting time until the next test suite execution is triggered which is either fixed ($|T| = 1$), randomized ($|T| = 2$) or iteration

dependent ($|T| \geq I$).

$$TS = \langle \mathcal{TC}, I, F, T \rangle \tag{4.2}$$

To illustrate the usage of a test suite, we build on the trusted package installation test case TC^{TPI} described in the previous section. As an example, we assume that the execution of TC^{TPI} is triggered randomly within a time interval of 60 minutes, i.e., $T = \langle 0, 3600 \rangle$. Furthermore, the test suite is consecutively executed for 3000 times, i.e., $I = 3000$, having an 15 minute offset between every execution, that is, $F = 900$. Consequently, we can describe the test suite example containing a single test case TC^{TPI} as follows:

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

Corresponding requirements Through combining test cases in an arbitrary manner, test suites enables *reuse of existing components of continuous tests*. This means that existent test cases can be reused with other test suites.

4.3.4 Workflow

A *workflow* determines the sequence of test suites which is executed over time. To that end, a workflow uses the results of the last test suite run to decide which test suite to execute next. A continuous test has exactly one workflow. The most basic example of a workflow always executes the same test suite, regardless of the result the last test suite run produced. Alternatively, a workflow may define to, e.g., execute a different test suite or terminate the test once a test suite run fails.

Recall that we aim to use continuous tests to reason about properties of a cloud service over time. A workflow addresses this goal by providing the necessary means for fine-grained modeling of test sequences. For example, a workflow can be defined which aims to estimate the downtime of a cloud service most accurately. This is leveraged in the example test in Section 5.2 to continuously test availability: This test's workflow initializes test execution with a test suite called *regular* which checks the availability of cloud service components randomly at least every 15 seconds and at most every 215 seconds. In case this regular test suite fails at some iteration because a cloud service component is not available, the workflow chooses a different test suite called *alert* to run next whose interval is smaller, that is, which is executed every 10 seconds. As long as the alert test suite fails, the workflow always selects the alert test suite to run next. Once the alert test suite passes, the workflow selects a test suite called *attentive* to run next. If the attentive test suite passes for five consecutive times, the workflow selects the regular test suite to run next, otherwise it runs the alert test suite again. It is important to note that all three test suites only differ with regard to the interval definition while test cases bound to these suites are identical.

We describe a workflow as a function $W : R \rightarrow \mathcal{TS}$ which takes as input the results of executed test suites R where each $r \in R$ is a 2-tuple of a test suite $TS^{\langle \mathcal{TC} \rangle}$ and a sequence of test suite's results Y after the i-th iteration, that is, $|Y| = i$. Recall the test suite example $TS^{\langle TPI \rangle}$ 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. The input provided to W for test suite $TS^{\langle TPI \rangle}$ after the third iteration is $r = \langle TS^{\langle TPI \rangle}, \langle \textit{passed, passed, failed} \rangle \rangle$. W then outputs the test suite $TS \in \mathcal{TS}$ 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.

Corresponding requirements A workflow can support *reusing existing components of a continuous test* because a particular workflow can be used for different tests. Consider, for example, the workflow to approximate downtimes of cloud service components described above (see also Section 5.2): This workflow can also be used to approximate the time a TLS configuration of a service endpoint has been vulnerable. Naturally, test cases bound to the *regular*, *alert* and *attentive* test suites have to be changed in order to test for vulnerable TLS configurations. However, the workflow approximating downtimes is agnostic to the test cases used and thus supports approximating the time a vulnerable configuration persisted in the same manner.

Furthermore, a workflow can be used to provide *self-adaptivity* of a continuous test to environment conditions which evolve over time. This application of a workflow is detailed in *Precondition as specialized test suites* of Section 4.3.6.

4.3.5 Test metrics

Continuous tests automatically and repeatedly produce evidence to support checking whether a cloud service complies with a set of controls over time. Thus, we need to interpret a sequence of test results in order to reason about cloud service properties over a period of time. To that end, we use *metrics* to compute *measurements* which are needed to evaluate statements over cloud services properties such as the availability of a cloud service's components is greater than 99.999% per day.

We describe a metric as a function $M : R \rightarrow U$ which takes as input results of test suite runs R and outputs measurements U. A metric can leverage 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.

Recall that test cases form the primitive of each test which use test oracles to determine the outcome of a test case, that is, whether a test cases passes or fails (see Section 4.3.2). Further, test suites combine test cases where each suite contains at least one test case. A test suite either passes or fails, it passes if all contained test cases pass (see Section 4.3.3). Any test metric used by a continuous test which strictly follows the building blocks defined in this chapter can therefore make use of the following two characteristics: First, a single test suite run (i.e., a single execution of a test suite as part of a continuous test) either passes or fails. As a consequence and second, a single test suite run passes or fails *at some point in time*. Based on these two characteristics, we propose four test metrics which are universally applicable to any continuous test which aims to support cloud service certification, independent of particular designs of test cases, test suites or workflow.

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

As Figure 4.3 shows, a basic test result is only returned after the execution of a test completed, that is, when a test suite run completed (tsr_i^e). This metric can be used to evaluate statements which only require to know if and how often a continuous test failed or passed. As an example, consider determining if and how often security groups assigned to a newly started virtual machine unexpectedly allow that these machines are publicly accessible through some blacklisted ports.

Failed-Passed-Sequence-Counter (fpsC) A continuous test continuously produces basic test results. A *Failed-Passed-Sequence* (fps) is a special type of sequence of basic test results: As shown in Figure 4.2, a fps starts with a failed test at t_i provided that the previous test at t_{i-1} passed. A fps ends with next occurrence of a passed test.

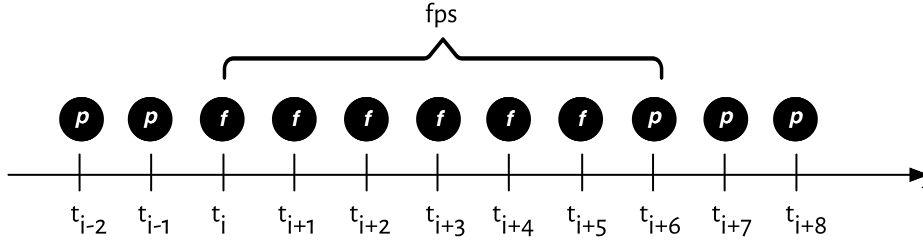


Figure 4.2: Failed-Passed-Sequence (fps) example based on basic test results (br)

Consider, for example, having observed the following sequence of basic test results of a continuous test: When trying to connect to a VM for eleven times in a row, the first two times the login succeeds (p). Then, for the next six times, the login fails (f) while for the remaining three times, the test passes again. The fps in this example is $fps^{11}_{SSH} = \langle f, f, f, f, f, f, p\rangle$.

The Failed-Passed-Sequence-Counter ($fpsC$) metric draws on this definition of fps. $fpsC$ counts the number of occurrences of fps within a sequence of basic test results $S_{br} = \langle br_1, br_2, \ldots, br_i\rangle$ which are produced during a continuous test. For example, consider Figure 4.2 which shows the following sequence of basic test results $\hat{S}_{br} = \langle p, p, f, f, f, f, f, p, p, p\rangle$. This sequence contains one fps, that is, $fpsC(\hat{S}_{br}) = 1$.

Failed-Passed-Sequence-Duration (fpsD) The Failed-Passed-Sequence-Duration ($fpsD$) metric builds on the definition of a Failed-Passed-sequence (fps) described in the previous paragraph. $fpsD$ takes a fps as input and measures the time between the first failed test of a fps and its last basic test result which – by definition – passes. This metric can be used to reason about properties over individual periods of time. This allows us to evaluate statements which contain time constraints. For example: A control derived from, e.g., *RB-21: Handling of vulnerabilities, malfunctions and errors – check of open vulnerabilities* of BSI C5 [31] may state that an incorrectly configured and thus insecure web server's TLS setup of a SaaS application has to be corrected within a certain amount of time, e.g., eight hours.

The definition of $fpsD$ has a subtle but important detail: We aim to measure the time difference between the first and the last test of a *Failed-Passed-Sequence*

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

An important insight at this point is that the first failed test f_i as well as the next passed test p_{i+j} each have duration, that is, both tests takes some time to complete and return a basic test result. Therefore, we have to choose whether a $fpsD$ *starts* at the start time or the end time of the first test f_i. Also, we have to decide whether a $fpsD$ *ends* at the start time or the end time of the last test p_{i+j}.

In order to properly define the bounds of a $fpsD$, we have to first understand how the different options may affect our metric. As an introductory example, consider Figure 4.3 which illustrates the definition of $fpsD$ using the start time tsr^s_i of the first failed test tsr_i and the end time tsr^e_{i+j} of the next passed test tsr_{i+j}. The duration of the first failed test and the last passed test are d_i and d_{i+j}, respectively.

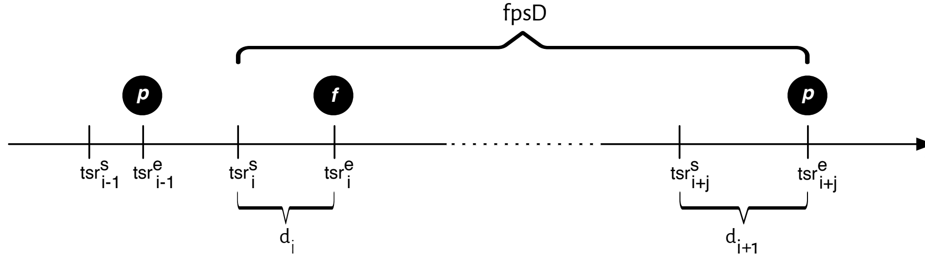


Figure 4.3: Continuously executed tests (tsr) with universal test metric $fpsD$

Yet the definition of $fpsD$ shown in Figure 4.3 has a severe disadvantage: The more time it takes to complete the last test tsr_{i+j}, the higher the proportion of d_{i+j} within the $fpsD$. Thus choosing tsr_i^s and tsr_{i+j}^e as bounds for $fpsD$ creates a dependency between the duration of tsr_{i+j} and the $fpsD$. In scenarios where high accuracy of $fpsD$ is required, e.g., to evaluate statements which contain relatively narrow time constraints, this dependency can render the metric $fpsD$ unsuited.

As mentioned in the introduction of this section, the metric $fpsD$ has to be applicable to any continuous test. Thus, when defining $fpsD$, we have to avoid dependencies of $fpsD$ from the duration a specific last test tsr_{i+j}. In order to arrive at a definition of $fpsD$ which is least dependent on tests' duration, we have to examine how variations in the duration of the first failed test (d_i) and the last passed test (d_{i+j}) affect $fpsD$.

The four options to define $fpsD$ are shown in Table 4.1. Consider, for example, *Option 3*: Here, we use the end of the first failed test (tsr_i^e) as start of the $fpsD$ and the end of the next passing test (tsr_{i+j}^e) as end of the $fpsD$. When choosing this option, variations of either the duration of the first test ($\Delta\ d_i$) or the last test (Δd_{i+j}) will affect the $fpsD$, i.e., lead to $\Delta fpsD$. Further, when the duration of both tests vary ($\Delta d_i \wedge \Delta d_{i+j}$), then this will also change $fpsD$, that is, $\Delta fpsD$. Note that in the corner case where variations in duration of both tests cancel each other out, i.e., $\Delta d_i = \Delta d_{i+j}$, $fpsD$ remains unaffected.

Table 4.1: Options to define Failed-Passed-Sequence-Duration ($fpsD$) if $|fps| > 2$

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

When inspecting Table 4.3, it becomes apparent that *Option 2* is the only definition of $fpsD$ which is unaffected by variations of duration of the first and the last test. Therefore, we choose to define the *start* of a $fpsD$ to be the start time of the first failed test tsr_i^s and for the *end* of a $fpsD$, we select the start time of the next passed test tsr_{i+j}^s.

It is important to note that the reasoning shown in Table 4.1 only applies to *Failed-Passed-Sequences* which contains more than two basic test results, that is, $|fps| > 2$. If a fps only contains two elements, i.e., $fps = \langle f_i, p_{i+1} \rangle$, then variations of the duration of the failing test f_i will impact on $fpsD$. Also, if $|fps| = 2$, then the $fpsD$ will be at least as long as it takes the failing test to complete. As a result, the time it takes to complete the first failing test also

constitutes the lower bound on how accurately we can reason about statement containing time constraints.

Finally, we note that the accuracy of a measurement computed by the test metric $fpsD$ heavily depends on the concrete implementation of a test. In Chapter 7, we analyze such timing related measurement errors of continuous test implementations in context of $fpsD$.

Cumulative-Failed-Passed-Sequence-Duration (cfpsD) This metric draws on the idea of the *Failed-Passed-Sequence-Duration* ($fpsD$) introduced in the previous paragraph. $cfpsD$ takes as input a sequence $\hat{S}_{fpsD}$ which contains any $fpsD$ observed during a continuous test, and returns their accumulated value. Naturally, $cfpsD$ may only be applied to $\hat{S}_{fpsD}$ if the sequence contains more than one $fpsD$. Otherwise, if $|\hat{S}_{fpsD}| = 1$, then $cfpsD = fpsD$.

The metric $cfpsD$ permits us to reason about the satisfaction of cloud service properties within a predefined period of time. Similar to the metric $fpsD$, we use $cfpsD$ to evaluate statements containing time constraints. In contrast to $fpsD$, $cfpsD$ allows us to evaluate statements whose time constraints refer to multiple property violation events observed within a particular period of time. As an example, consider control *RB-02 Capacity management – monitoring* of the Cloud Computing Compliance Controls Catalogue (BSI C5) [31] which requires the cloud providers to comply with promised service level agreements (SLAs). Let's consider a SLA which defines that the total yearly downtime of a cloud service must not surpass five minutes. During the period of a year, the cloud service experiences multiple, timely separated downtime events which a continuous test detects. The metric $fpsD$ can evaluate statements which contain a single downtime to, e.g., not last longer than 60 seconds. In contrast, $cfpsD$ considers a period of time, e.g., a year, and summarizes over any $fpsD$ observed to evaluate statements referring to all downtime events during the defined period.

Similar to $fpsD$, the accuracy of a measurement computed by the test metric $cfpsD$ is determined by the concrete implementation of a test. Chapter 7 experimentally investigates measurement errors of concrete test implementations with regard to $cfpsD$.

Corresponding requirements Test metrics are the means within our framework which allow us to model the *production of evidence* which can be used to evaluate whether controls of a certificate are satisfied. More specifically, test metrics take as input test suite results and thus allow us to define models how to evaluate cloud service properties over time.

4.3.6 Preconditions

Naively executing tests is prone to produce false negative test results, e.g., testing if the TLS configuration of a service endpoint is secure from a remote host may fail not because of a vulnerable configuration but because the component cannot be reached due to a network error. Computing test metrics based on false negative test results will lead to erroneous metrics and thus incorrect evaluations of statement over the cloud service. Therefore, the assumptions made about the environment of the cloud service under test, i.e., *preconditions*, need also to be tested. Only with satisfied preconditions can we use test results to compute test metrics.

Our framework provides two options to model preconditions. 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 designed 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 executed the main test suite if all preconditions have passed. Therefore, preconditions can be used to control the workflow of a continuous test, allowing to self-adapt, i.e., select and execute test suite according to environmental conditions discovered at runtime.

Figure 4.4 shows an extract of an example continuous test that uses a specialized test suite 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.

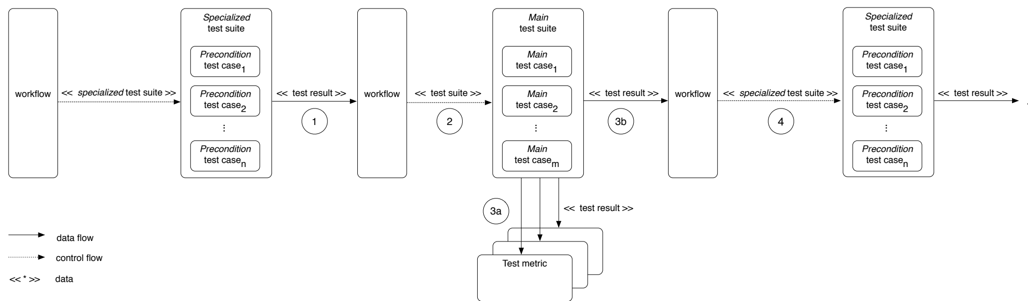


Figure 4.4: Extract of an example continuous test using a specialized test suite to test preconditions before executing the main test suite

Consider, as an example, that a 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 the duration of that upload is measured. One example of a precondition for this test to execute correctly is that the VM is reachable via SSH. In order to evaluate whether this precondition holds, we can 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 4.3.3, 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 preconditions as test cases and binding them to the main test suites. Figure 4.5 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 4.3.3), 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 controlling the workflow and have it self-adapt accordingly. 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.

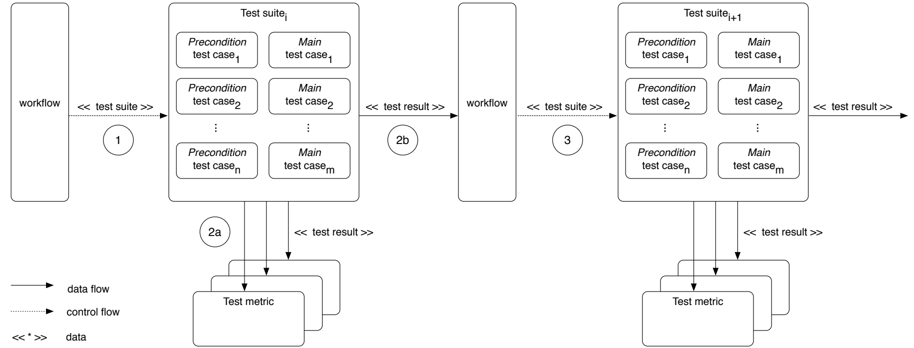


Figure 4.5: Extract of an example continuous test using precondition test cases as part of the main test suites

Modeling preconditions as part of main test suites has the advantage that we can enforce that execution of testing preconditions and execution of the main test suite are triggered concurrently. Moreover, note that the ordering numbers – which are required for definition test cases (see Section 4.3.2) – allow for fine-grained pairing of precondition test cases and main test cases within the test suite.

Since we are handling failed preconditions during metric computation, we have to also investigate how *discarding* test results impacts on test metrics. Regarding the Basic-Result-Counter (brC), we will avoid adding failed test to brC^F whose preconditions were not satisfied. Thus preconditions impact on the test metric brC by removing false negative test results which is the very purpose of precondition testing.

However, the effects of testing preconditions can be detrimental to the accuracy of any metric which is based on Failed-Passed-Sequence (fps). This includes Failed-Passed-Sequence-Counter ($fpsC$), Failed-Passes-Sequence-Duration ($fpsD$), as well as Cumulative-Failed-Passed-Sequence-Duration ($cpfsD$).

Let's assume that we have observed the following sequence of basic test results

$$S_{br} = \langle p_i, f_{i+1}, f_{i+2}, f_{i+3}, f_{i+4}, p_{i+5} \rangle.$$

If all preconditions tested to produce the basic test results of S_{br} succeed, then S_{br} contains one fps with five elements, that is,

$$fps_s = \langle f_{i+1}, f_{i+2}, f_{i+3}, f_{i+4}, p_{i+5} \rangle.$$

Lets now consider that the preconditions of basic test result f_{i+3} are not satisfied. As a result, f_{i+3} is therefore discarded during metric computation. After the correction, we get

$$\hat{S}_{br} = \langle p_i, f_{i+1}, f_{i+2}, f_{i+4}, p_{i+5} \rangle \text{ which contains } \widehat{fps}_s = \langle f_{i+1}, f_{i+2}, f_{i+4}, p_{i+5} \rangle.$$

Now consider applying the metric $fpsC$ which counts the occurrence of fps: The count of fps remains unaffected by discarding f_{i+3}, that is, $fpsC(\hat{S}_{br}) = fpsC(S_{br})$. However, removing f_{i+3} does affect the computation of metric $fpsD$. The reason is that the duration of the fps is computed on the time difference between p_{i+5} and f_{i+1} which implicitly includes f_{i+3}. Consequently, by including the incorrect test result f_{i+3} in the fps, we incorrectly add time to the $fpsD$. This error is passed onto the test metric $cfpsD$ since it accumulates any $fpsD$ observed during a continuous test.

In order to correctly compute the $fpsD$ test metric, we have to deduct the duration during which we know that the preconditions were not satisfied. To that end, we have to inspect any fps for test results whose preconditions failed (tsr^-). Note that a fps may contain multiple tests whose preconditions failed which occur in direct succession. In order to describe failed preconditions, we define a *Preconditions-Failed-Sequence*

$$pfs = \langle tsr^+_{i-1}, tsr^-_i, tsr^-_{i+1}, tsr^-_{i+2}, \ldots, tsr^+_{i+j} \rangle$$

whose first (tsr^+_i) and last element (tsr^+_{i+j}) are test results whose preconditions are satisfied (tsr^+). Any element of a pfs that follows the first (tsr^+_i) and precedes the last (tsr^+_{i+j}) is a test result whose precondition failed (tsr^-).

Recall that we aim to find the interval during which preconditions of successive tests were not satisfied. Figure 4.6 shows how we compute the duration of a pfs, that is, $pfsD$: It is the difference between the start of the last test tsr^s_{i+j} and the end of the first test tsr^e_{i-1}.

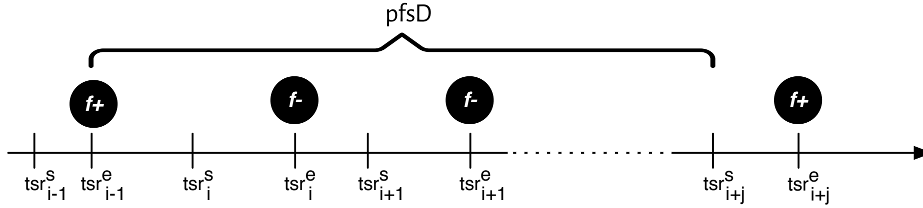


Figure 4.6: Bounds of a *Preconditions-Failed-Sequence-Duration* ($pfsD$)

A fps can contain multiple pfs, at most $k = |fps|\ div\ 2$. For example, $fps_s = \langle f_{i+1}, f_{i+2}, f_{i+3}, f_{i+4}, p_{i+5} \rangle$ can contain at most $5\ div\ 2 = 2\ pfs$. In general, we have to consider multiple $pfsD$ when correcting $fpsD$ as follows:

$$\widehat{fpsD} = fpsD - \sum_{i=0}^{k} pfsD_i.$$

Finally, there is the following corner cases to consider: The precondition of the first failing test of a Failed-Passed-Sequence (fps) is not satisfied. Considering Figure 4.6, this means that preconditions of test result at tsr_{i-1} are not fulfilled. In this case, the definition of the duration of a pfs has to be changed slightly, it is then the difference between the start of the last test tsr^s_{i+j} and the start of the first test tsr^s_{i-1}.

Corresponding requirements The option *Precondition as specialized test suite* of the building block *Precondition* draws on the workflow in order to adapt to evolving environment conditions of a continuous test, thereby addressing the requirement *self-adaptivity*. This requirement is also addressed when using the option *Preconditions as part of main test suites*: In this case, preconditions are used in combination with test metrics where test metrics are corrected in case precondition testing fails.

4.4 Example implementation of the framework

Figure 4.7 shows *Clouditor*[55], a set of tools supporting design and deployment of continuous assurance techniques. The Clouditor toolbox consists of five main components: *Engine*, *explorer*, *simulator*, *evaluator* and *dashboard*. Hereafter, we outline Clouditor's *engine* and its *dashboard*. Further details on the remaining components can be found in [245].

It is important to note that Clouditor's *engine* implements the framework described in Section 4.3 and thus depicts a core results of this thesis. However, Clouditor's *dashboard* is *not* a result of this thesis but has been created by Clouditor's development team following the implementation of the engine. The dashboard is included in this Chapter for illustration purposes only.

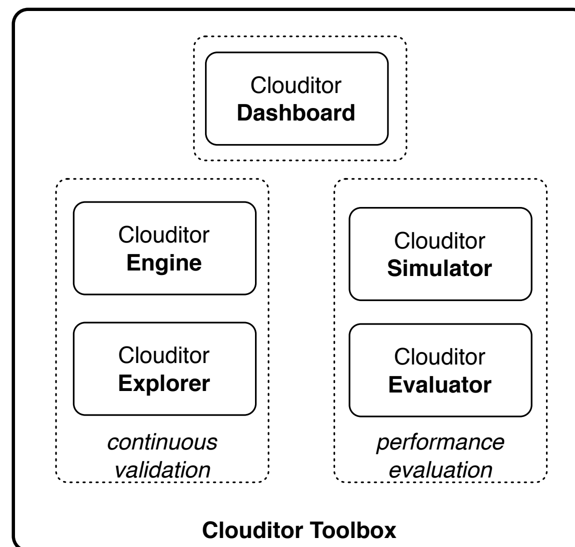


Figure 4.7: Components of the Clouditor toolbox

4.4.1 Clouditor's engine

Clouditor's *engine* is a prototype which is developed in Java following the design of the building blocks described in the previous section. Using the engine, novel continuous tests can be build by implementing test cases – possibly integrating other existing, external tools such as Nmap[56] – which are then combined to test suites. Aside from some basic workflow templates applicable to a wide range of test scenarios (e.g., *SimpleRepetitionWorkflow* which always executes the same test suite), the engine supports implementation of custom workflows deciding which test suite to execute next. Similarly, the engine provides the set of universal test metrics introduced in Section 4.3.5 (i.e., Basic-Result-Counter (brC), Failed-Passed-Sequence-Counter ($fpsC$), Failed-Passed-Sequence-Duration ($fpsD$), and Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$)). However, the engine also supports implementation of custom metrics, for example, to compute the current strength of TLS cipher suites supported by a cloud service component's endpoint based on the last test suite results.

[55] `https://clouditor.io` [Accessed: 2018-12-13]
[56] `https://nmap.org/` [Accessed: 2018-12-13]

Clouditor's engine supports concurrent execution of multiple continuous tests. Provided that all custom classes required for a specific set of continuous tests were added to the engine, the desired tests can be defined using YAML configuration files (see also Section 6.2.4) where a each test is defined by a single configuration file. Results produced by the engine, that is, test suite run results (which include the results of test cases bound to the test suite), measurements (i.e., results of the test metric computation) as well as the test definitions used to configure tests are persisted in a MongoDB[57] database. Clouditor's engine is currently deployed as a containerized application running on Kubernetes[58] within the laboratory environment of Fraunhofer AISEC[59].

4.4.2 Clouditor's dashboard

Clouditor's *dashboard* is a web application which provides a management console as well as visualizes the results produced by Clouditor's engine. Depending on the test metrics defined by the continuous tests which are executed, different visualization components, such as time series graphs, burn-up charts or maps are supported.

Figure 4.8 shows a screen shot of the dashboard's landing page. The top three tiles show three properties which are currently subject to continuous tests: *Availability*, *Geolocation*, and *TLS Configuration*. The components of the cloud service which are tested are shown in the tile on the left bottom: Three virtual machines (i.e., *VM1*, *VM2* and *VM3*), a *LoadBalancer* and an *OpenStackAccount*. Further, the tile on the right bottom of the landing page shows a summary of the test results produced during the last 24 hours by all continuous tests currently running. Each box represent one hour and is colored green if all test suite results during that hour indicate success, otherwise the box is colored red (i.e., at least one test suite run during that hour failed). Lastly, boxes are colored from left to right as time passes where gray boxes indicate that no results during that (upcoming) hour have been produced yet.

Selecting one particular test from the right bottom tile shown in Figure 4.8 opens a detailed view on the results this test produced. Figure 4.9 shows a screen shot of this view for the continuous test of the availability of *VM1*. On the left top tile, the test results during the last 24 hours are shown where each box represents one minute. Again, red boxes indicate failure of at least one test suite run during the respective minute while green ones indicate passed test suite runs. Further, the bottom right tile shows a burn up chart which illustrates the cumulative downtime as indicated by the test results. This test metric computation follows the Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) definition introduced in Section 4.3.5. Lastly, the right top tile shows the test metric *Availability* which, in this case, is defined as the ratio between the cumulative downtime and the period of time since the continuous test started (more details on this test metric computation can be found in Section 5.2.3.4).

---

[57]`https://www.mongodb.com/` [Accessed: 2018-12-13]
[58]`https://kubernetes.io/` [Accessed: 2018-12-13]
[59]`https://www.aisec.fraunhofer.de/en.html` [Accessed: 2018-12-13]

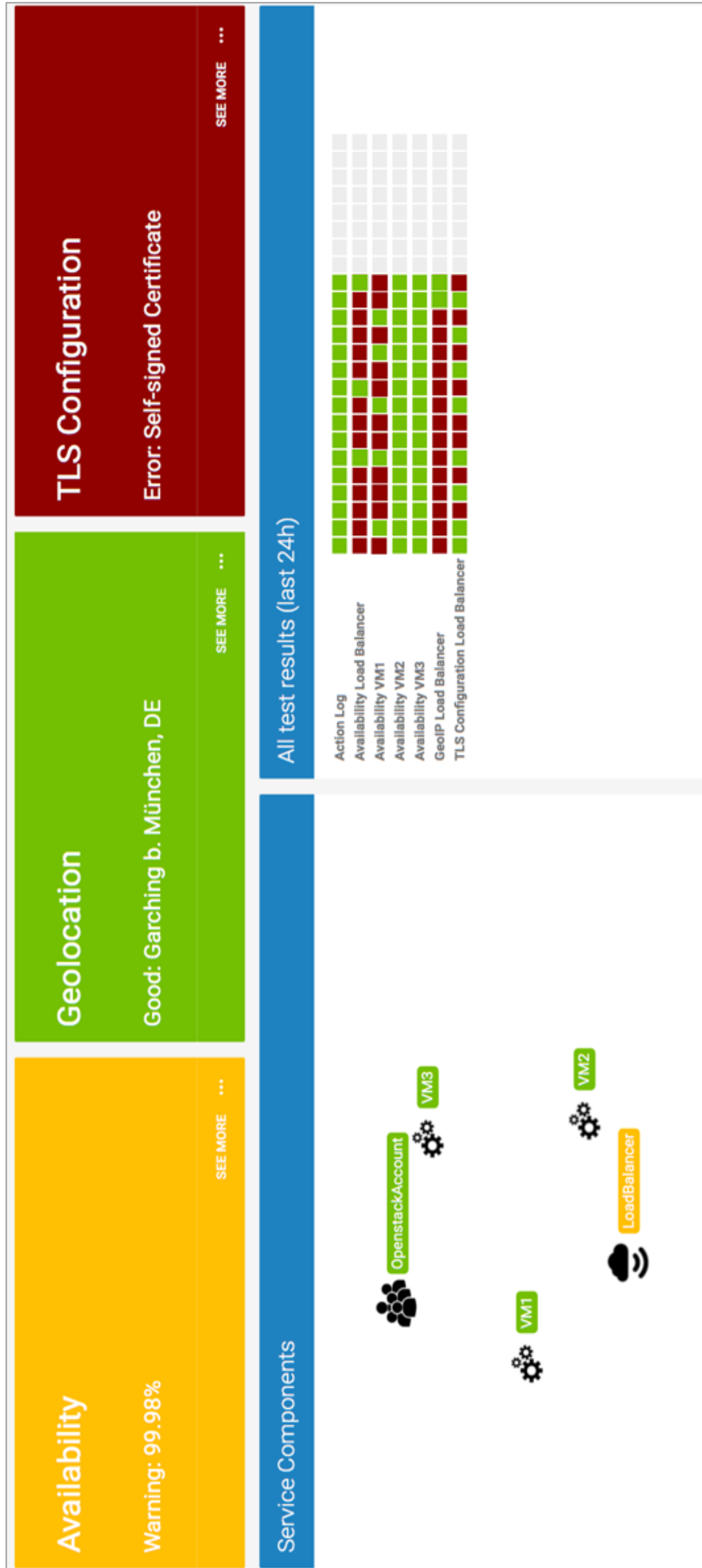


Figure 4.8: Landing page of Clouditor's dashboard

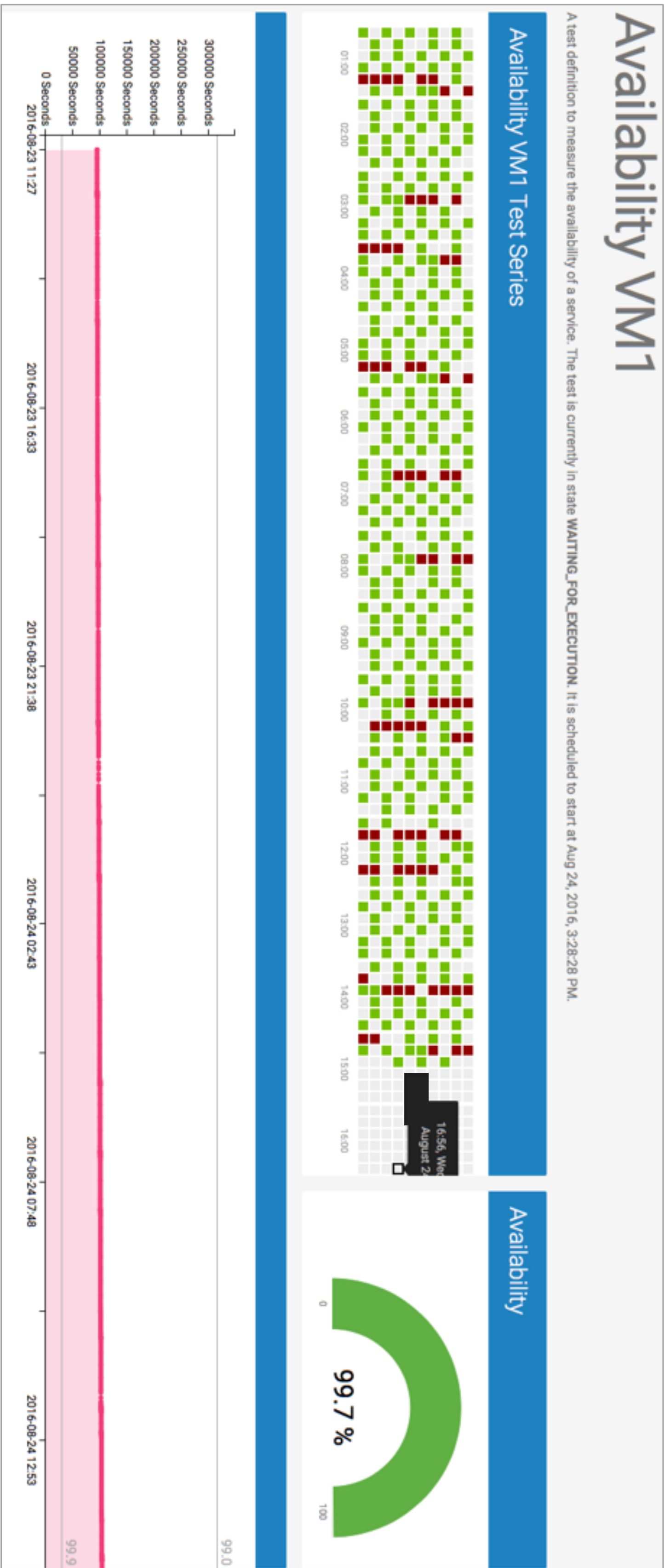


Figure 4.9: Different visualizations of test suite results and test metrics of example continuous test *Availability VM1*

4.5 Summary and discussion

In this chapter, we introduced a framework to support continuous test-based certification of cloud services. The main objective of this framework is:

> *Support design of tests which permit to continuously evaluate properties of existing cloud services where test results serve as evidence to validate that the service satisfies a set of controls.*

Based on this overall goal, we then elicited the requirements which the framework has to meet to achieve the following subgoals:

- Production of evidence,
- extensibility,
- flexible integration with existing infrastructures of cloud services,
- independence of cloud service model,
- reusability of continuous test components,
- integration of existing test tools, as well as
- self-adaptivity.

Then we presented the building blocks of the framework which are guided by the above requirements. Those building blocks are:

- Test cases,
- test suites,
- workflow,
- test metrics, and
- preconditions.

The design of the framework corresponds adequately with the identified requirements: *Test cases* are very flexible primitives allowing to *integrate existing tools* such as SQLMap, i.e., execute this tool and use the returned results. Test cases also support adding novel tests (*extensibility*) which can evaluate properties of SaaS, PaaS, or IaaS, or combinations thereof (*independence of cloud service model*). Also, test cases can be design to *flexibly integrate with existing cloud service infrastructures*, either tightly integrated with privileged access or in a minimally invasive manner, that is, only requiring minimal changes to the existing service and having non-privileged access. *Test suites* combine test cases which enables *reuse of existing components of continuous tests*, in this context, reusing existent test cases within different test suites.

Further, the *workflow* of a continuous test determines which test suite is executed next. Therefore, a workflow meets two of the identified requirements: First, it allows to *reuse existing continuous test components*, i.e., here reusing existent test suites. Second, it allows a continuous test to adapt based on changing environment conditions during execution

(precondition testing with specialized test suites, see Section 4.3.6) as well as based on observed test results (*self-adaptivity*).

Test metrics take as input the results produced by test suites and permit us to evaluate cloud service properties over time. Therefore, test metrics are the constructs within our framework which allow us to model the *production of evidence* which is needed to check satisfaction of a certificate's controls. Further, test metrics may correct produced evidence in case preconditions are not satisfied (preconditions as part of main test suites, see Section 4.3.6), thereby also supporting the required *self-adaptivity* of continuous tests.

Moreover, we propose the following four test metrics which are universally applicable to any continuous test aiming to support cloud service certification, independent of particular designs of test cases, test suites or workflow:

- Basic-Result-Counter (brC),
- Failed-Passed-Sequence-Counter ($fpsC$),
- Failed-Passed-Sequence-Duration ($fpsD$), and
- Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$).

Lastly, we outlined one example implementation of the framework's building blocks. This prototype constitutes the *engine* component of *Clouditor*, a set of tools supporting design and deployment of continuous assurance techniques.

One drawback of our framework lies in neglecting non-functional requirements which need to be considered when designing continuous tests which are to be productively deployed. While continuous tests seek to increase customer's trust in and transparency of cloud service, such mechanisms can also leak critical information which can be used by adversaries. For example, Santos et al. [246] point out that such information can be used to trace vulnerabilities of a cloud infrastructure. As an example, consider that a continuous test checks whether a SaaS application is vulnerable to SQL Injections (SQLI) which can, e.g., support validation of CCM's control *TVM-02: Vulnerability & Patch Management*. It is evident that leaking test results which indicate a SQLI vulnerability – possibly even containing detailed information on the type of vulnerability found – can simplify the steps an attacker has to take to attack the service. Therefore, it is essential to ensure that the system which implements continuous tests in a productive setting is trustworthy as well [120].

Aside from proposing a suitable security model for continuous test implementations, the required performance of productive continuous tests cannot be ignored. After all, a prevalent attribute of cloud services is scalability of available resources (for further details see Section 2.1). However, neither designing and implementing a most efficient continuous tests nor keeping their memory and storage usage as low as possible is in focus of this work. Nevertheless, the practical applicability of continuous tests hinges on their capability to handle the elasticity of cloud services' resources. Thus discussing potential limitations of our framework in context of productively deployed continuous tests and proposing possible remedies is subject to future work.

Finally, as already pointed out in the introduction of this Chapter, our framework's focus lies on providing technical building blocks to design continuous tests. Yet it does not specify a process how to manually or automatically conceive test design candidates for a concrete cloud service instance. It is thus left to future work to answer the question *'Given a particular cloud service instance, which continuous tests are feasible?'*

Furthermore, the framework does not aim at providing a method to rigorously derive test designs from control descriptions. This implies that our framework does not address questions such as *'Given a particular control, which test designs are most suitable?'*, *'How many controls of a control catalog such as CSA CCM [22] can be continuously evaluated using tests?'* or *'Which additional costs does continuously evaluating controls of CSA CCM incur?'* since the answers require having established a rigorous link between controls and test designs.

Chapter 5

Example continuous test scenarios

This chapter presents five example test scenarios which are designed based on the framework introduced in the previous chapter. The primary goal of these test scenarios is to demonstrate how our proposed framework can support continuous test-based certification of cloud services.

There are various standards and guidelines available (see Section 2.3.3) whose controls we can use to derive example test scenarios. Naturally, our goal is to demonstrate that our framework can support a wide range of cloud-specific controls. However, it is outside the scope of this thesis to provide a set of test scenarios which is most complete, that is, covers as many controls of standards and guidelines possibly relevant to cloud services. This would imply that ideally any control of any available certification scheme had to be considered during the selection process. Even though this is a feasible task, it would involve considerable, ongoing manual effort and is not in focus of this work. Fortunately, organizations such as the Cloud Security Alliance (CSA) and German Federal Office for Information Security (BSI) have made it part of their core business to compose lists of controls specific to cloud services which we can employ. We use controls of the following three control catalogs as a basis for each of our example test scenarios:

- Cloud Computing Compliance Controls Catalogue (BSI C5) [31] of the German Federal Office for Information Security (BSI),
- Cloud Control Matrix (CCM) [22] provided by the Cloud Security Alliance (CSA), and
- ISO/IEC 27001:2013 [24] published by the International Organization for Standardization (ISO).

We select BSI C5 because only shortly after its publication in the middle of 2016, the IaaS market leader Amazon Web Services (AWS) already adopted this standard[60]. Furthermore, CSA's CCM and the ISO/IEC 27001:2013 are on the *Cloud Certification Schemes List (CCSL)*[61] which was compiled during the implementation of the *European Cloud Strategy* [11], an initiative of the European Commission. While ISO/IEC 27001:2013 is not specific to cloud services, it is a very popular standard which finds wide application within various

[60]https://aws.amazon.com/compliance/bsi-c5/ [Accessed: 2018-12-13]

[61]https://resilience.enisa.europa.eu/cloud-computing-certification [Accessed: 2018-12-13]

industries[62], including companies providing cloud services such as AWS [248].

The focus of this chapter – as already pointed out above – lies on demonstrating how our framework supports continuous cloud service certification. This requires deriving conclusions about our framework which go beyond the scope of merely demonstrating the feasibility of the five example test scenarios presented in Sections 5.2 – 5.6. Therefore, we define general characteristics of continuous test scenarios (Section 5.1) which we then apply to each selected scenario. This allows us to reason about the general applicability of our framework. The design of the continuous test of each test scenario follows the notation introduced in Section 4.3. Also, each scenario includes a description of the implementation as well as experimental results of deploying the respective continuous test.

Note that test designs, configurations as well as experimentally induced states of the service under test which are presented in this chapter are examples demonstrating test-based evidence production using our framework. Since – as already pointed out in the previous chapter – there is no rigorous approach available to derive the most suited test designs on the basis of a control's description, these tests may be designed and configured differently, depending on their concrete use cases and the interpretations of the respective controls. Lastly, it is important to point out that we assume the cloud service providers within our example scenarios to *not* behave maliciously. This means that a provider does *not* attempt to cheat on the tests in order to manipulate the results in his favor. We investigate test-based certification of an adversarial cloud provider in Chapter 8. Note that parts of this chapter have been published in [127], [244], [249] and [250].

5.1 General characteristics of continuous test scenarios

In this section, we identify four general characteristics of continuous test scenarios. We use these characteristics to describe each of the following test scenarios (Sections 5.2 – 5.6). As a result, we are not limited to our five example test scenarios but can draw further conclusions about the applicability of our framework to support continuous test-based cloud service certification. The characteristics of continuous test scenarios are introduced hereafter.

1. *Plausible link to cloud-specific controls:* The evidence which is produced by testing a cloud service property has to be suited to establish a plausible link to one or more cloud-specific controls according to which a cloud service is to be certified. If there is no plausible link between the property and a cloud-specific control, then we cannot derive an example test scenario from that property because testing the property will not support certification of the cloud service. Naturally, what constitutes a *plausible link* is the result of subjective assessment and may be disputed.

   Describing this characteristic entails to first identify cloud-specific controls. To that end, ideally any control of any certificate should be considered during the selection process to identify those which are cloud-specific. Even though this is a feasible task, it involves considerable, ongoing manual effort. Fortunately, there are organizations such as the Cloud Security Alliance (CSA) who have made it part of their core business to compose lists of controls specific to cloud services which we can employ.

   Once potential cloud-specific controls have been identified, we have to provide a plausible argument which links these controls to the continuous test of a cloud service

[62] The ISO Survey of Management System Standard Certifications states that – as of 31st December 2015 – 27536 certificates according to the ISO/IEC 27001:2013 standard have been issued [247].

property. As mentioned above, this part may give room for debate, however, through explicitly describing the presumed plausible link, we make the underlying reasoning transparent, comprehensible and replicable.

2. *Supported cloud service models:* As pointed out in the previous paragraph, a continuous test scenario aims at demonstrating how our framework tests a cloud service property and thus supports cloud service certification. Therefore, it has to be possible to describe the relation between the property which is tested and one or more cloud service models to which a cloud service under test possessing this property belongs to.

   As introduced in Section 2.1.1, there are three different cloud service models, namely *SaaS*, *PaaS*, and *IaaS*. The relation between a property and cloud service model indicates that the property is actually cloud-specific. Further, this relation maps the property to one or more cloud service models. This, in turn, allows us to describe that our framework supports continuous test-based certification of each cloud service model.

3. *Implementation independence:* While it has to be possible to relate a property to service models of the cloud service under test, the property ideally is not specific to a cloud service implementation. If a cloud service uses, for example, some proprietary technology to encrypt data in transit, then a test which continuously checks if this communication channel is secure does not necessarily apply to other cloud services. Therefore: If a cloud service property of a test scenario is implementation-specific, then the conclusions which we can draw from the scenario are limited to only a few instances of cloud services.

4. *Minimally invasive integration:* Recall the integration levels of tests introduced in Section 4.2.3: Some cloud service properties can be tested without changing configuration or composition of the service (*non-invasive integration*), while others require configuration alterations (*minimally invasive integration*) or even adding components or changing code of the service to be tested (*invasive integration*).

   The obvious advantage of invasive tests lies in the privileged access this integration level provides which may translate to more comprehensive and expressive evidence produced by invasive tests: Installing an agent on any virtual machine of a particular service, for example, allows to test if only authorized personnel can access these machines via SSH. However, adding new applications to cloud service components or applying code changes increases the probability of introducing vulnerabilities and other flaws which lead to increased risks of the cloud service not operating securely and reliably. Furthermore, it is reasonable to expect that invasive integration of tests with existing cloud services incurs higher costs of integration and operation (since at least parts of the test have to be deployed as part of the service's infrastructure) than non-invasive and minimally invasive integration. Lastly, when neglecting tests that use platform specific API bindings to conduct checks on the service's control plane (i.e., platform level), then most non-invasive and minimally invasive tests will be applicable to cloud services' components regardless of the underlying platform of the cloud service provider. For example, testing the strength of TLS cipher suites provided by a cloud service's public endpoint is agnostic to the underlying infrastructure.

   With regard to characterizing continuous test scenarios to reason about the general applicability of our framework, the above comparison of integration levels leads to the

conclusion that integration of tests within the scenarios should ideally be non-invasive or at last minimally invasive. The reason for this is that invasive integration would implicitly assume that cloud service providers are willing to tolerate increased risk and costs which, in turn, restricts reasoning about the framework's general applicability to this set of providers. Furthermore, in most cases, invasive tests will be specific to the platform of the provider whose service components are under test. Such platform dependencies would, again, restrict conclusions about the framework's general applicability.

5.2 Continuously testing availability

In this section, we present a test which aims at continuously checking whether a cloud service is available. We begin with defining the property *availability* in the context of a cloud service and outline the test scenario. Then we explain the general characteristics of the test scenario (Section 5.2.2) and present the test design (Section 5.2.3). Finally, we present experimental results of using our test to evaluate the availability of IaaS provided by OpenStack (Section 5.2.4). Parts of this section have been published in [244].

5.2.1 Property description and overview of test scenario

First, we have to define what the property *availability* of a cloud services means. Tanenbaum and Steen [251] describe *availability* of a system as the probability that it is operating correctly and as expected by its users at any time. Therefore, a *highly* available cloud service is one which is highly likely to be working as expected by its users at any moment.

From a customer perspective, using cloud services is a form of outsourcing resources and applications, that is, not deploying and maintaining these resources since this is the responsibility of the cloud service provider. However, usually customers are not willing to tolerate downtimes of these provided resources because outages can lead to immediate, detrimental effects on their business model, e.g., loss of productivity. This constitutes the reason why availability is frequently included as a service level objective as part of a service level agreement (SLA). As we will see in Section 5.2.2.1, SLA satisfaction, in turn, is part of controls defined by standards such as BSI C5.

Unavailable cloud services can, for example, be caused by specific types of attacks, so-called *Denial-of-Service (DoS)* attacks or by human error. The latter case could be observed on February 28th of 2017: The Simple Storage Service (S3) of AWS was disrupted as a result of an incorrectly issued command by one of the administrative personnel, also affecting other services such as AWS EC2 (IaaS) and AWS Lambda (serverless computation, PaaS) [252]. This example illustrates two important issues: First, due to intrinsic characteristic of a cloud service to facilitate scalability through quick provisioning of resources, even small operational errors can have major impacts. Second, considering a large cloud service provider such as AWS having a large number of customers, such outages may directly affect a whole range of cloud services and thus negatively impact on the respective service customers.

In order to continuously test the availability of a cloud service, we propose to test the availability of its composing resources. We refer to this test as CT^{AV} which repeatedly measures round trip times on the Internet Layer using ICMP packets and on the Transport

Layer using TCP segments to determine whether a cloud service component, e.g., a virtual machine, is available. Therefore, our interpretation of the availability of cloud service components is driven by the reachability and responsiveness of its interfaces using ICMP and TCP. Note that our framework can also be used to design application-specific availability tests, e.g., calling the RESTful API of a PaaS and checking if returned objects are correct and provided in time.

In summary, the continuous test works as follows (a detailed description of the test will be provided in Section 5.2.3): In *regular mode*, CT^{AV} randomly waits between 15 and 215 seconds and then tests if a cloud service component is reachable via ICMP and TCP. In case this test fails, then CT^{AV} switches to the *alert mode* in which it tests statically every 10 seconds to more accurately approximate the downtime. Once a cloud service component is reachable again, it slowly returns to *regular mode*. Finally, CT^{AV} uses the produced test results to calculate a test metric which aims at estimating the total downtime of the cloud service within a specified period of time, e.g., within one year.

5.2.2 General characteristics of the test scenario

This section describes the general characteristics of the scenario to test the availability of cloud service components.

5.2.2.1 Supported cloud-specific controls

This sections aims at presenting examples of cloud-specific controls which are related to the availability of cloud service components. We begin with the control *RB-02 Capacity management – monitoring* of the Cloud Computing Compliance Controls Catalogue (BSI C5) [31]. 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.".*

The continuous test CT^{AV} can support the certification of a cloud service according to this control since it repeatedly tests whether cloud service components are available. In order to ensure that contractual agreements and SLAs are satisfied, CT^{AV} computes a test metric quantifying the downtime within a specified period of time.

Further, the BSI provide a complementary document [253] in which they map the controls of BSI C5 to existing international standards which, among others, include CSA's Cloud Control Matrix (CCM) [22] upon which the CSA certificate is based [23]. In this document, the BSI maps the control *RB-02* of BSI C5 to *IVS-04* of the CSA's CCM:

> *"The availability, quality, and adequate capacity and resources shall be planned, prepared, and measured to deliver the required system performance in accordance with legal, statutory, and regulatory compliance obligations. Projections of future capacity requirements shall be made to mitigate the risk of system overload.".*

As we can see, this control describes that availability of a cloud service shall be measured to be compliant. Thus, the mere operation of our test CT^{AV} would already support a certification where this control has to be satisfied.

CSA's CCM provides a mapping of their controls to those of other guidelines and certification schemes which are not necessarily specific to cloud services. The above introduced control *IVS-04*, for example, links to the control *A.12.1.3 Capacity management* of ISO/IEC 27001:2013 [24]. Note that – as already mentioned above – the standard ISO/IEC 27001:2013 is not cloud-specific. However, control *IVS-04* which is part of the CCM's control domain *Application & Interface Security* as well as *RB-02* of BSI C5 link to *A.12.1.3* of ISO/IEC 27001:2013, thus we can treat it as applicable to cloud services.

5.2.2.2 Supported cloud service models

The continuous test CT^{AV} uses round trip times which are repeatedly measured on the Internet Layer using ICMP packets and on the Transport Layer using TCP segments to check if a cloud service components such as virtual machine are available. Therefore, we interpret the availability of cloud service component based on whether its interfaces are reachable as well as responsive using ICMP and TCP.

This test can be used with any of the three cloud service models or combinations thereof: Consider, as an example, a virtual machine as one instance of IaaS. In order to determine whether this IaaS is available, we can use its publicly exposed IP address or hostname and repeatedly evaluate the round trip time on the Internet Layer. Further, as connections to virtual machines are usually established via SSH on port 22, we can evaluate the response time on the Transport Layer. As an example for PaaS, consider a platform service such as CLOUD SQL[63], a database service provided by the Google Cloud Platform. Similar to the case of IaaS, we can use the publicly exposed IP address or hostname and the TCP port of the database service to repeatedly measure and evaluate network delay on the Internet and Transport Layer, thus determining whether it is available.

Lastly, this continuous test can also be used to check whether a SaaS application is available. Since SaaS applications are typically accessed through browsers, we can use the hostname which points to web site offering the SaaS application and port 80 (HTTP) or port 443 (SSL/TLS) to check if is available, that is, if it can be reached on the Internet and Transport Layer.

We note that our test tells us whether an end point is reachable and responding as expected. Application-specific availability measurement, that is, determining whether a particular application operates correctly at a certain point in time requires some protocol adaptions to add application-specific semantics, e.g., calling a RESTful API and checking whether the returned object is correct and provided in time. Our framework can be used to design test which conduct application-specific availability measurements. To that end, test cases have to be implemented which check the availability on the application level.

5.2.2.3 Implementation independence

Since we use delay measurements on the Internet Layer and on the Transport Layer, our continuous test does not depend on the specific implementation of a SaaS, PaaS, or IaaS instance. For example, the test is applicable regardless of the operating system a virtual

[63] `https://cloud.google.com/sql/` [Accessed: 2018-12-13]

machine uses (IaaS) or what kind of web server is used by a SaaS application. Naturally, the only indispensable prerequisite for CT^{AV} is that the cloud service components under test can be reached over an IP network and that they use some Application Layer protocol such as HTTP or SSH which use TCP on the Transport Layer.

5.2.2.4 Minimally invasive integration

Although it is possible to deploy our continuous test CT^{AV} as part of the cloud service under test, that is, adding the implementation of the test as component to the service, such an invasive change in composition of the service is not required. Also, testing the availability of components from a host which is part of the cloud service's infrastructure would only provide limited information if the service is working as a user remotely accessing service's components expects. However, it depends on the cloud service model what kind of configuration alterations may be required in order to allow for CT^{AV} to function properly. These ramifications are explained hereafter.

If the continuous test is executed on a remote host and the cloud service component is publicly reachable, e.g., a web server of a SaaS application, then no change to configuration or composition of the cloud service is needed. Yet if the component whose availability we aim to check is a virtual machine (IaaS) to which access is restricted by a firewall, then changing firewall rules in order to permit the test to reach the instance is required.

5.2.3 Test design

In this section, we describe the test CT^{AV} according to the framework introduced in the Chapter 4. Note that the configuration of CT^{AV} serves as an example and it may vary depending on the use case of the test as well as on the interpretation of the respective controls whose validation it supports.

5.2.3.1 Test cases

The availability test consists of two test cases TC^{ICMP} and TC^{TCP}. The former one is defined as follows:

$$
\begin{aligned}
TC^{ICMP} =& \langle E, L, O, N \rangle \\
=& \langle \langle \mathit{measure_delta_ICMP_Echo_Request_to_ICMP_Echo_Reply_packets} \rangle, \\
& \langle \langle \mathit{host,\ packet_count} \rangle \rangle, \\
& \langle \mathit{assert_rtt_avg} < \$^{ICMP}_{avg},\ \mathit{assert_rtt_sd} < \$^{ICMP}_{sd} \rangle, 1 \rangle
\end{aligned}
$$

The test case TC^{ICMP} uses one procedure (E), i.e., it measures the time delta between sending ICMP Echo Request and receiving ICMP Time Reply packets. To that end, the input parameters (L) $host$ and $packet_count$ are required, specifying either IP address or hostname of the cloud service component's endpoint and number of packets to send for one test, respectively. Since a single test case execution obtains multiple successive probes at a time (we assume that $packet_count > 1$), each test case result is actually a distribution. We define two oracles (O) to evaluate the average and standard deviation of the test case result using the assertions $assert_rtt_avg < \$_{avg}$ and $assert_rtt_sd < \$_{sd}$. Thus an instance of TC^{ICMP} passes if the returned round trip time (rrt) satisfies both of the assertions. Note that the placeholder $\$^{ICMP}_{avg}$ and $\$^{ICMP}_{sd}$ have to be assigned the desired expected values

in milliseconds, e.g., 20ms and 10ms, respectively. These expected values depend on the environment where CT^{AV} is deployed and have to either be based on previously observed, i.e., historic values or educated guesses what constitutes an expected delay. Lastly, the ordering number (N) of TC^{ICMP} is 1 which means that the test case is executed first once execution of the test suite is triggered.

Furthermore, the test case TC^{TCP} is defined as follows:

$$\begin{aligned} TC^{TCP} =& \langle E, L, O, N \rangle \\ =& \langle \langle \textit{measure_delta_SYN_to_SYN-ACK_TCP_segment} \rangle, \\ & \langle \langle \textit{host, probe_count, port} \rangle \rangle, \\ & \langle \textit{assert_average_response_time} < \$^{TCP}_{avg}, \textit{assert_max_response_time} < \$^{TCP}_{max} \rangle, 1 \rangle \end{aligned}$$

Similar to TC^{ICMP}, this test case has also only one procedure (E) which measures the time delta between sending a SYN TCP segment to and receiving a SYN-ACK TCP segment from the endpoint of the cloud service component. The required input parameters (L) are either IP address or hostname ($host$), a TCP port ($port$) and the number of probes ($probe_count$). Analogous to TC^{ICMP}, a single execution of TC^{TCP} results in a distribution of measured delays. In order to evaluate the results produced by the test case, we define two oracles (O) $\textit{assert_average_response_time} < \$^{TCP}_{avg}$ and $\textit{assert_max_response_time} < \$^{TCP}_{max}$ which assert average and maximum of the measured response times of the cloud service component's endpoint. That means that an instance of TC^{TCP} fails if either or both the assertions do not hold. The placeholders $\$^{TCP}_{avg}$ and $\$^{TCP}_{sd}$ are assigned the expected desired values, e.g., 75ms and 100ms, respectively. Similar to the test case TC^{ICMP} described above, these expected values depend on the deployment of CT^{AV} and can be based on, e.g., historic values. Finally, the ordering number of TC^{TCP} is also 1 which means that TC^{ICMP} and TC^{TCP} will be executed concurrently.

5.2.3.2 Test suites

The continuous availability test consists of three test suites. The first one is defined as follows:

$$\begin{aligned} TS^{\langle ICMP,TCP \rangle}_{\text{regular}} =& \langle \mathcal{TC}, I, F, T \rangle \\ =& \langle \langle TC^{ICMP}, TC^{TCP} \rangle, +\infty, 15, \langle 0, 200 \rangle \rangle. \end{aligned}$$

The test suite $TS^{\langle ICMP,TCP \rangle}_{\text{regular}}$ binds both test cases ($\mathcal{TC}$) TC^{ICMP} and TC^{TCP}. Further, the number of successive iterations of $TS^{\langle ICMP,TCP \rangle}_{\text{regular}}$ is set to infinity which is indicated by setting iterations (I) to $+\infty$. This means that successive executions of this test suite can potentially be triggered infinitely often unless the workflow decides – based on produced test results – to execute a different suite. Further, each execution of each test suite run is triggered randomly in the interval (T) $\langle 0, 200 \rangle$ seconds after the last test suite run completed. Lastly, there is a 15 seconds offset added to the interval between successive test suite executions (F).

Furthermore, the second test suite is defined as follows:

$$\begin{aligned} TS^{\langle ICMP,TCP \rangle}_{\text{alert}} =& \langle \mathcal{TC}, I, F, T \rangle \\ TS^{\langle ICMP,TCP \rangle}_{\text{alert}} =& \langle \langle TC^{ICMP}, TC^{TCP} \rangle, +\infty, 0, \langle 10 \rangle \rangle. \end{aligned}$$

This test suite binds the test cases TC^{ICMP} and TC^{TCP} as well. Its number of successive execution (I) is also set to $+\infty$, i.e., it has no maximal number of successive iterations and

thus may be triggered infinitely often. Furthermore, the execution of this test suite is triggered statically every 10 seconds (T) after the previous test suite run completed, having *no* fixed offset (F).

The definition of the third test suite is:

$$TS_{\text{attentive}}^{\langle ICMP,TCP\rangle} = \langle \mathcal{TC}, I, F, T\rangle$$
$$TS_{\text{attentive}}^{\langle ICMP,TCP\rangle} = \langle\langle TC^{ICMP}, TC^{TCP}\rangle, 5, 5, \langle 25, 35, 45, 55, 65\rangle\rangle.$$

Also, this test suite binds both test cases TC^{ICMP} and TC^{TCP}. Yet its number of successive executions (I) is limited to five, each of which having its individual waiting time between successive executions (T). That is the reason why the number of iterations holds exactly five elements, i.e., $I = |T| = 5$. Thus, for example, before running the first instance of this test suite, we wait 25 seconds, before running the second, we wait 35 seconds and so forth. Finally, there is a fixed offset of 5 seconds between each test suite run (F).

5.2.3.3 Workflow

The workflow we use aims at estimating the downtime of the cloud service component starting from the first point of detection. Figure 5.1 shows how this workflow executes the three test suites which we introduced in the previous paragraph: The test starts with the test suite TS_{regular}. If this test suite fails at some iteration, then the next test suite which is executed is TS_{alert}. This test suite is executed repeatedly as long as it fails. If TS_{alert} passes at some point, then the test suite executed next is $TS_{\text{attentive}}$. If $TS_{\text{attentive}}$ passes for five consecutive times, then $TS_{\text{attentive}}$ is stopped and TS_{regular} is started. If one of $TS_{\text{attentive}}$ five iterations fail, then $TS_{\text{attentive}}$ is stopped and TS_{alert} is started. Once TS_{alert} passes again, $TS_{\text{attentive}}$ is run next and so forth.

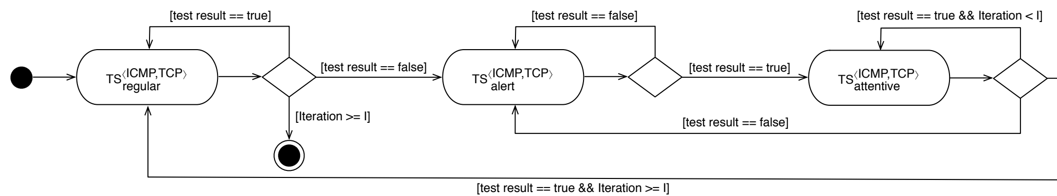


Figure 5.1: Workflow of continuously testing resource availability (CT^{AV})

As mentioned above, the goal of this workflow is to approximate the downtime of the cloud service component. Figure 5.2 illustrates how the different configurations of the three test suites support this goal: While the test suite TS_{regular} randomly waits $[15, 215]$ seconds after the previous completed, the test suite TS_{alert} statically executes ten seconds after the previous one completed. Thus, once an instance of TS_{regular} failed, we repeatedly test with shorter intervals which allows to more accurately approximate the downtime of the cloud service component. Once the cloud service component is available again, the test suite $TS_{\text{attentive}}$ is executed which has to pass exactly five times where the interval between each execution increases by ten seconds. The idea here is that if a downtime event just occurred, then it is more likely that another downtime event follows shortly afterwards.

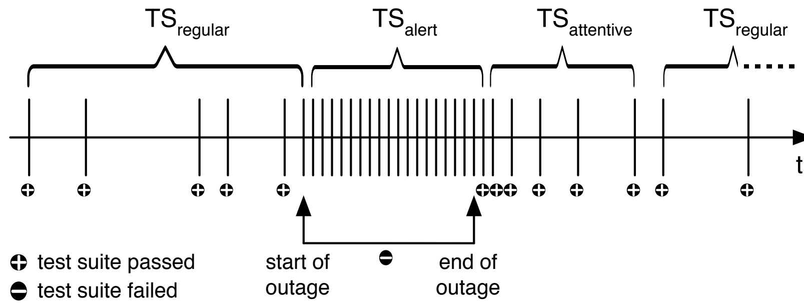


Figure 5.2: Extract of a test sequence to estimate downtime of a cloud service component

5.2.3.4 Test metric

The resource availability test aims at determining the total downtime of a cloud service component within a specified period of time, e.g., a year. Within that period, multiple downtimes may be detected by the continuous test which have to be accumulated to provide a global value.

In order to define the test metrics of the resource availability test, we draw on the universal test metrics *Failed-Passed-Sequence-Duration* ($fpsD$) and *Cumulative-Failed-Passed-Sequence-Duration* ($cpfsD$) introduced in Section 4.3.5: We use $fpsD$ to determine the duration of any singular downtime event of a cloud service component. Recall that a $fpsD$ is the duration of a *Failed-Passed-Sequence* (fps) which is defined by the difference between the start of the first failed test suite run and the start of the next test suite run that passes. In context of the resource availability test, this translates to successive failure of any of the three test suites described in the previous paragraph. More specifically, the first element of a fps is either produced by a failed TS_{regular} or by a failed $TS_{\text{attentive}}$. Moreover, the last element of a fps is always produced by a passing TS_{alert}.

Having observed multiple singular downtimes, we sum over the corresponding $fpsD$ to obtain $cpfsD$ which holds the total downtime within the specific period where CT^{AV} is operating. Since we are not aiming at checking the downtime but the availability of a cloud service component, two last steps are required to compute final test metric: First, we simply compute the ratio between the total downtime and the period of time CTD during which CT^{AV} was deployed:

$$cfpsD_{\%}^{CT^{AV}} = \frac{cfpsD^{CT^{AV}}}{CTD}.$$

This gives us the percentage of downtime which when deducted from 1, leads us to the final test metric which indicates the availability of a cloud service component:

$$M_{\%}^{CT^{AV}} = 1 - cfpsD_{\%}^{CT^{AV}}.$$

Recall that in the introduction of this section, we adopted the definition of availability provided by Tanenbaum and Van Steen [251], thus treating availability as the probability of a cloud service's component to work correctly at any given time. In order to conform with this definition, we regard values calculated for $M_{\%}^{CT^{RA}}$ as estimates, that is, as expected values for the availability of a cloud service's component. We note that this estimation method is basic and more advanced methods exist which, for example, are capable of considering the sequence of observed test results. Yet it is not the focus of this work to propose a novel method to estimate the availability of cloud service's components. However, our framework

may accommodate other, more sophisticated test metrics which is, after all, exactly what this test scenario aims to demonstrate.

5.2.4 Implementation and experiment

This section describes the implementation of the continuous test CT^{AV} as well as present experimental results.

5.2.4.1 Environment and setup

Hereafter, the main components of the experiment are outlined.

Cloud service component under test As an example for a cloud service component under test, we choose an instance of a virtual machine provided by OpenStack Mitaka[64]. This instance is running a Ubuntu 15.10 Server and is equipped with 2 VCPUs and 4 GB RAM and 40 GB volume (disk). Furthermore, the instance is associated with a publicly reachable IP address and its security group are configured to allow for traffic via ICMP, via TCP on port 22 and via SSH from and to the host where the continuous test is deployed (see paragraph *Deployment of continuous test* below).

Implementation of test cases The test case TC^{ICMP} is implemented using Ping[65], a common network diagnostic tool which can be used to measure response times on the IP Layer. Furthermore, the test case TC^{TCP} is implemented using Nping[66]. This tool can be used to measure response times on the TCP Layer. Recall that both test cases are bound to test suite $TS^{\langle ICMP,TCP \rangle}$ and are executed concurrently. Once both test cases complete, the output of either tool is parsed and it is evaluated if the test cases passed. Since the assertions depend on the environment of the deployment, we have to define them at this point:

- $\$^{ICMP}_{avg} = 100$ms and $\$^{ICMP}_{sd} = 75$ms, as well as
- $\$^{TCP}_{avg} = 75$ms and $\$^{TCP}_{max} = 100$ms.

Deployment of continuous test The test CT^{AV} is deployed on a different host than the cloud service component under test. This host is attached to a different network than cloud service component under test. However, both hosts are located inside the same building which leads to relatively low delays on the IP and TCP Layer (< 3 ms) observed previous to conducting the experiment.

Inducing downtime events We trigger 1000 temporary downtime events of our cloud service component under test. To that end, we use OpenStack4J[67], a Java library which permits controlling OpenStack, including the management of virtual machines. Each downtime event starts with pausing the cloud service component, i.e., the virtual machine, and ends with unpausing it. Furthermore, each downtime event lasts at least 60 seconds plus selecting

[64] `https://www.openstack.org/software/mitaka/` [Accessed: 2018-12-13]
[65] `https://linux.die.net/man/8/ping` [Accessed: 2018-12-13]
[66] `https://nmap.org/nping/` [Accessed: 2018-12-13]
[67] `http://openstack4j.com/` [Accessed: 2018-12-13]

another [0,60] seconds at random. The interval between consecutive downtime events is set to last at least 120 seconds plus an additional [0,60] seconds which are selected randomly.

The accumulated duration of all downtime events observed during the experiment is 91438.67 seconds (≈ 25 hours 23 minutes 59 seconds). Further, the mean duration of each downtime event is 91.49 seconds with a standard deviation of 17.87 seconds.

5.2.4.2 Experimental results

In total, the whole experiment took ≈67.5 hours, i.e., starting from the start of the first test until the end of the last test. Table 5.1 shows test statistics and test results which continuous test CT^{AV} produced: The total number of tests, i.e., the total number of test suite executions is 6118. The execution of a test suite of CT^{AV} took 12.76 seconds on average with a standard deviation of 4.73 seconds.

Out of the 1000 downtime events, CT^{AV} correctly detected 989 or 98.9% ($fpsC$). Furthermore, each $fpsD$ on average estimates that the duration of a correctly detected downtime event is 70.73 seconds. Since each downtime event actually lasted 91.49 seconds on average, CT^{AV} on average underestimates a downtime event by 20.76 seconds. Consequently, when considering the accumulated duration of detected events ($cfpsD$), then CT^{AV} only detects 69956.6 seconds of the accumulated duration of downtime events (91438.67 seconds) which equals ≈76.51%.

Finally, when considering the interval between the start of the first test and end of the last test as the period of time during which CT^{AV} was deployed, then we can compute the test metric which indicates the availability of our cloud service component:

$$cfpsD_{\%}^{CT^{RA}} = \frac{cfpsD^{CT^{RA}}}{CTD} = \frac{69956.6}{242998.35} \approx 0.2879$$

leading to

$$M_{\%}^{CT^{RA}} = 1 - cfpsD_{\%}^{CT^{RA}} = 1 - 0.2879 = 0.7121.$$

Thus we can state that the availability measured by the continuous test CT^{AV} was 71.21%. Note that – since CT^{AV} underestimates detected the downtime events – the actual availability as provided by the accumulated duration of downtime events is lower, i.e., ≈ 62.37%.

5.3 Continuously testing location

In this scenario, we propose a test which continuously validates the geographical location of cloud service components. We start with defining what the property *location* means in context of cloud services and outline the test scenario. Then we investigate the general characteristics of the test scenario (Section 5.3.2) and describe the test design (Section 5.3.3). Lastly, we present implementation as well as experimental results of using our continuous test to validate the locations of IaaS instances provided by Amazon Web Services (AWS) (Section 5.3.4). Parts of this section have been published in [250].

5.3.1 Property description and overview of test scenario

The location of a cloud service component is *valid* if it is in agreement with the expected location of that component. The first question at this point is why cloud service components are likely to change the location at which they are hosted. As described in Subsection

Table 5.1: Test statistics and results of continuous test CT^{AV}

Test statistics	number of tests	6118
	mean duration test (sec)	12.76
	sd duration test (sec)	4.73
	min duration test (sec)	9.04
	max duration test (sec)	19.24
fpsC	correctly detected events	989
	correctly detected events (%)	98.9
fpsD	mean duration detected events (sec)	70.73
	sd duration detected events (sec)	27.29
	min duration of detected events (sec)	19.08
	max duration of detected events (sec)	136.41
cfpsD	accumulated duration detected events	69956.6
	accumulated duration detected events (%)	76.51

2.1.1, components of a cloud service are usually virtualized, including virtualization of networks, physical servers, and storage. Migrating these virtual components from one geographical location to another is a standard feature which most cloud service provider offer their customers. As an example, consider migrating IaaS, i.e., virtual machines from one geographic location to another one. Amazon Web Service (AWS) and Google CloudPlatform provide standard features to make images of custom virtual machines and copy them to another geographical region where they then can be launched[68][69]. Yet migrating cloud services is not confined to migrating virtual machines: AWS, for example, also supports migration of PaaS applications such as its Relational Database Service (AWS RDS) where – also as a standard feature – customers may migrate entire database clusters from one geographical location to another[70].

Naturally, when testing from a remote host, we can only reach the cloud service component's endpoint. In the example of IaaS, this is usually a publicly reachable IP address or public hostname which we can connect to using SSH. Similarly, in the example of PaaS, this endpoint can be some publicly reachable hostname which is part of the base URL of a HTTP-based RESTful API. Thus, a central question at this point is: Assuming that cloud service components are provided by remote hosts, what means do we have at hand to determine their location?

Various geolocation techniques aiming at determining *unknown* locations of Internet hosts use network delay measured on different layers of the TCP/IP protocol suite and topology information, i.e., the path taken to a specific host (e.g., [158][149][159][160] [161]). One fundamental limitation of these approaches is that they may only be able to determine the location of a proxy server but not the actual location of the Internet host [158][254][154][147]. In our case, this translates to locating the physical servers running the proxy instead of the

[68] `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/CopyingAMIs.html` [Accessed: 2018-12-13]

[69] `https://cloud.google.com/compute/docs/instances/moving-instance-across-zones` [Accessed: 2018-12-13]

[70] `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_CopySnapshot.html` [Accessed: 2018-12-13]

physical server on which the cloud service components are deployed, e.g., virtual machines.

Yet even if we were to neglect this limitation, the conclusions drawn about a cloud service component's location are confined to locating the physical server on which the component is running. A natural extension to locating Internet hosts – which is outside of the scope of our continuous test – is to determine the geographical location of *data* stored by a cloud. Determining that a cloud provider is storing data at some expected geographical location is referred to as *data sovereignty* which requires simultaneously establishing the geographical location of the server on a network and proving that the data is actually stored at that location [150][151]. Still, this approach does not cover locating any other copies of the data stored on a different server. Without having full control of the network, tracking any such copies is a different (and hard) problem [150].

Our continuous test CT^{LV} uses *location classification* which treats potential locations, i.e., geographical areas where cloud service components can be hosted as classes. A *classifier* is a supervised learning algorithm [255] which – in our continuous test scenario – uses some characteristics of a cloud service component to predict to which class it belongs, that is, at which location the component is hosted. An important difference to the above discussed geolocation techniques is that our approach requires any potential location to be *known*.

We use network delay and topology information to predict, i.e., classify the location of a cloud service component. Furthermore, our test shares one other important assumption with the above mentioned geolocation techniques: They require that there are trusted *landmarks*, that is, geographical locations which are known and from which network delay and topology information are measured.

Recall that we are aiming at continuously validating the location of cloud service components using classifiers. This entails a challenge which – although recognized by several approaches, e.g., [159][146] – remains unaddressed: Network delay as well as topology information are subject to changes over time as described in, e.g., [256][257]. Therefore, we have to take network changes over time into account as otherwise these changes can render our location predictions inaccurate. Such time-dependent learning problems are referred to as learning under *concept drift* [258][259].

In order to continuously validate the location of cloud service components, our continuous test CT^{LV} implements a process which continuously collects, predicts and updates a classifier. Note that not every deviation from previous observations necessarily indicates a concept drift, some may result from momentary or short-termed anomalies. Distinguishing anomalies from permanent network changes is crucial as we otherwise update our classifier with flawed data.

The workings of this continuous test can be summarized as follows (a detailed description of the test will be provided in Section 5.3.3): Given a set of targets, i.e., hostnames of cloud service components, CT^{LV} initially collects delay measurements on the IP and TCP Layer as well as topology information. Using these data points, a classifier is trained. Thereafter, new measurements using the targets are conducted which are then used to predict the locations of the cloud service components. Before locations are predicted, however, potential outliers are filtered and only those newly collected probes are fed to the classifier which are considered normal. Thereafter, the initial data set is augmented using the newly collected probes and the classifier is updated, i.e., retrained. Then, again, delay measurements and topology information using the targets are collected, outliers are filtered, their location is predicted and so forth. The result which is produced by CT^{LV} indicates whether a cloud service component is located at its expected location by comparing the expected location with the predicted one.

5.3.2 General characteristics of the test scenario

This section describes the general characteristics of the scenario to continuously test the location of cloud service components.

5.3.2.1 Supported cloud-specific controls

The control *UP-02 Jurisdiction and data storage, processing and backup locations* of BSI C5 [31] requires that

> "*[...] Data of the cloud customer shall only be processed, stored and backed up outside the contractually agreed locations only with the prior express written consent of the cloud customer.".*

This control demands that customer's data shall only be processed within contractually agreed location which, in our terminology, are valid locations. Thus, our continuous test CT^{LV} can support certification of a cloud service according to *UP-02* by continuously checking whether cloud service components still reside at contractually agreed location. The control *RB-03 management – data location* of BSI C5 is even more specific, stating that

> *"The cloud customer is able to determine the locations (city/country) of the data processing and storage including data backups.".*

BSI C5 explicitly links this control to the before presented *UP-02*, considering it supplemental. In order to fully understand how CT^{LV} can support certification of a cloud service according to this control, we have to shed light on a subtle detail which stems from this control's ambiguous use of the term *determine*: Either it refers to determining a previously *unknown* location or it requires determining *known* location of a cloud service's components where data is processed. As already mentioned above, the continuous test CT^{LV} uses a supervised learning algorithm, a classifier, to predict the location of IaaS which requires any potential location to be *known*.

The BSI also provides a complementary document [253] in which they map the controls of C5 to existing international standards which, among others, include CSA's CCM [22] and ISO/IEC 27001:2013 [24]. It turns out that – according to the BSI cross-referencing – neither of them contains a control which explicitly addresses the geographical location of a cloud service's components. However, both the CCM and ISO/IEC 27001:2013 do contain controls which demand compliance of a cloud service with regulatory obligations. For example, *AIS-01 Application & Interface Security Application Security* of the CCM requires that

> *"Applications and programming interfaces (APIs) shall be designed, developed, deployed, and tested in accordance with leading industry standards (e.g., OWASP for web applications) and adhere to applicable legal, statutory, or regulatory compliance obligations.".*

Further, control *A.18.1.4 Privacy and protection of personally identifiable information* of the ISO/IEC 27001:2013 can be considered relevant in this context. The European Union (EU) Data Protection Directive[71] is one example of a set of regulations where the location of cloud services' components can become important. It restricts personal information from flowing

[71] Note that in the course of preparing this thesis, the General Data Protection Regulation (GDPR) [260] superseded the Data Protection Directive.

from EU member states to any other country whose laws do not have an *adequate level of protection* [261]. Another example are the Australian Privacy Principles (APP) [262] which do not permit personal information flowing to a foreign country unless those country's laws are *substantially similar* to APP. Therefore, continuously validating the location of a cloud service's components using CT^{LV} can support certification of a cloud service according to controls which demand compliance with regulatory obligations.

5.3.2.2 Supported cloud service models

The continuous test CT^{LV} uses network delay measured on the Internet and Transport Layer as well as topology information to classify the location of a cloud service component. More specifically, on the *Internet layer*, we use the public IP address of the target host to ping it, i.e., measure the time delta between sending ICMP Echo Request and receiving ICMP Echo Reply packets. Also, we measure the time delta between sending ICMP Echo Request packets and receiving ICMP Time Exceeded packets for every hop until the target host is reached. In order to characterize the route taken to a target, we count the number of known and unknown intermediate routers. Furthermore, on the *Transport layer*, we measure the time delta between sending a SYN and receiving a SYN-ACK TCP segment from the target host on a particular port.

CT^{LV} can be used to validate the geographical location of IaaS, i.e., virtual machines hosted on remote servers. Similar to continuously testing the availability of a cloud service component (see Section 5.2), we can make use of a VM's publicly exposed IP address or hostname to collect measurements on the Internet Layer. Further, in order to measure delay on the Transport Layer, we can send SYN TCP segments to port 22 of the VM, the standard SSH port. Naturally, this implies that any virtual machines which constitutes a component of a cloud service is reachable from the landmark from which network delays are measured and topology information is gathered.

Using CT^{LV} to validate the location of SaaS and PaaS applications' components is only possible to a limited extent. The reason for this is that customers interact with SaaS and PaaS applications on a different level of abstraction. Although it is still feasible to use the public hostnames which are part of URLs used by of a SaaS or PaaS application to collect network delay and topology information, the conclusions which can be drawn based on these measurements are limited to endpoint, e.g., the location of the SaaS application's web server or load balancer. Other components of the application are intentionally hidden from the customer's point of view which also renders their location transparent. Naturally, the underlying components of a SaaS or PaaS application could be made accessible, thereby removing the abstraction and allowing to validate their location. However, breaking the abstraction in such a way leads to a scenario which is equivalent to continuously testing IaaS as discussed above because we would then validate the infrastructure components of the SaaS or PaaS application.

5.3.2.3 Implementation independence

Analogous to continuously testing availability of a cloud service's components (see Section 5.2), through measuring delay and topology information on the Internet and Transport Layer, our continuous test CT^{LV} is agnostic to the operating system of a virtual machine or applications running on that VM. The only requirement is that the VM can be reached over an IP network and uses some TCP-based protocol.

5.3.2.4 Minimally invasive integration

The design of the test CT^{LV} does not require to add the implementation of the test as a component to the cloud service. Hence, no change to the composition of the infrastructure of the cloud service components we seek to validate is needed. However, similar to the availability test CT^{AV}, repeatedly measuring the network delay on the Internet and Transport Layer as well as topology information requires that the IaaS, i.e., the virtual machines can be reached by and respond to ICMP packets and TCP segments. Therefore, in order for CT^{LV} to be able to work, potential access restrictions to the VMs have to be adapted to allow measurement from the landmark.

5.3.3 Test design

In this section, we describe the test CT^{LV} according to the framework introduced in the Chapter 4. Note that configured parameters of CT^{LV} (e.g., test suite iterations) are example values which may differ depending on the use case the test is used within.

5.3.3.1 Test cases

The first test case of CT^{LV} is defined as follows:

$$\begin{aligned} TC^{IPD} =& \langle E, L, O, N \rangle \\ =& \langle \langle \mathit{measure_delta_ICMP_Echo_Request_to_ICMP_Time_Reply_packets}, \\ & \mathit{measure_delta_ICMP_Echo_Request_to_ICMP_Time_Exceeded_packets}, \\ & \mathit{count_hops} \rangle, \\ & \langle \langle \mathit{host, packet_count} \rangle, \langle \mathit{host}, \mathit{packet_count} \rangle, \langle \rangle \rangle, \\ & \langle \mathit{assert_successful_ICMP_probe} \rangle, 1 \rangle. \end{aligned}$$

The test case TC^{IPD} has three procedures (E): The first one uses the IP address or hostname ($host$) of the cloud service component's endpoint as input parameter (L) to ping it, i.e., measure the time delta between sending ICMP Echo Request and receiving ICMP Time Reply packets. The second procedure also uses the host address as input to measure the time delta between sending ICMP Echo Request packets and receiving ICMP Time Exceeded packets for every hop until the target host is reached. A single data point is obtained by executing multiple successive delay measurements at a time ($packet_count$), e.g., 20. Thus, each delay measurement is actually a distribution which we describe by its max^{IP}, min^{IP}, average (avg^{IP}), standard deviation (sd^{IP}), and $last^{IP}$ packet's delay. The third procedure counts the number of known and unknown intermediate routers, thereby characterizing a route's topology taken to a target. An instance of TC^{IPD} successfully passes (O) if no error occurred during measurement, i.e., all four statistics are correctly returned and can be used for further processing. Lastly, the ordering number (N) of the test case is 1.

The next test case is defined as follows:

$$\begin{aligned} TC^{TCPD} =& \langle E, L, O, N \rangle \\ =& \langle \langle \mathit{measure_delta_SYN_to_SYN\text{-}ACK_TCP_segment} \rangle, \\ & \langle \langle \mathit{host, probe_count, port} \rangle \rangle, \\ & \langle \mathit{assert_successful_TCP_probe} \rangle, 1 \rangle \end{aligned}$$

TC^{TCPD} uses a single procedure (E) to measure the time delta between sending a SYN and receiving a SYN-ACK TCP segment from the target host on a particular port. To that end, it requires the host address (*host*), *port*, as well as the number of probes per test case run (*probe_count*) as input parameters (L). Analogous to TC^{IPD}, a single execution of TC^{TCPD} results in multiple successive probes whose distribution is described by max^{TCP}, min^{TCP}, and average (avg^{TCP}). Similar to TC^{IPD}, an instance of TC^{TCPD} passes (O) if all three statistics are returned correctly, ready to be further processed.

The third test case of CT^{LV} is responsible for training a classifier. It is defined as follows:

$$\begin{aligned} TC^{TRN} =& \langle E, L, O, N \rangle \\ =& \langle \langle \textit{train_classifier} \rangle, \\ & \langle \langle \textit{classification_algorithm, training_data} \rangle \rangle, \\ & \langle \textit{assert_trainingError} < \hat{\varepsilon} \rangle, 1 \rangle. \end{aligned}$$

TC^{TRN} consists of one procedure (E) which uses a set of labeled training data to train a classifier. To that end, we have to select a supervised learning algorithm, e.g., K-nearest neighbor [263] [264], support vector machines (SVM) [265], decision tree [266] or random forest [267], and provide suitable training data as input (L). The training data consists of delay measurements on the Internet and Transport Layer as well as topology information which are provided by the test case results of TC^{IPD} and TC^{TCPD}, respectively.

At this point, it is important to note that these delay and topology measurements have to be represented by a suitable data structure, i.e., a feature vector in order to be processed by the classifier as input. Therefore, we transform the measurements into a ten-dimensional feature vector which is shown in Figure 5.3: It includes the average over any descriptive statistic obtained from the delay measurements on the Internet layer, the known and unknown hop count, and the descriptive statistics of the delay measurements of the Transport layer.

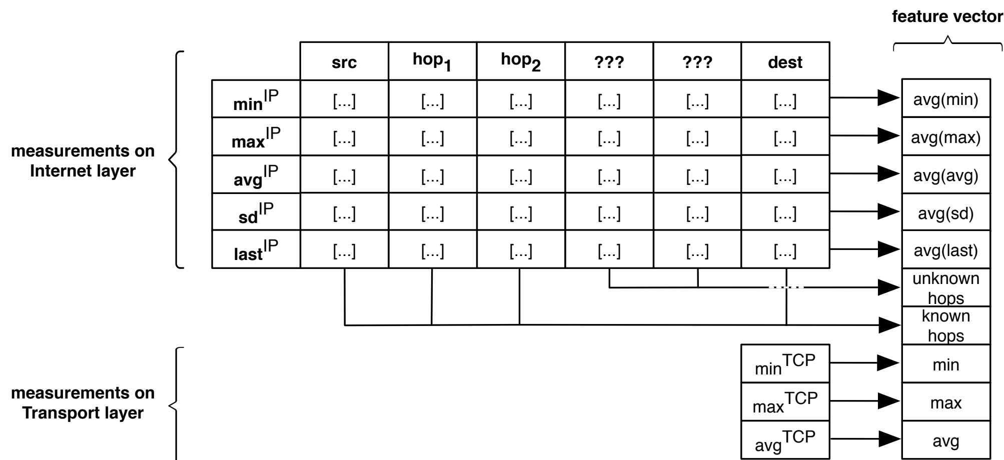


Figure 5.3: Feature vector derived from measurements on Internet and Transport layer provided by TC^{IPD} and TC^{TCPD}

Now the remaining question is on what condition an instance of TC^{TRN} passes. The oracle (O) which we define ties the answer to this question to the required performance of the classifier: We choose the *training error* ε, that is, the proportion of overall incorrectly

classified locations observed when using the trained model[72] to classify locations of training set to describe the performance of the classifier. We define the expected performance $\hat{\varepsilon}$ as an upper bound of the observed training error: If the observed training error ε is lower than $\hat{\varepsilon}$, then TC^{TRN} passes.

The fourth test case TC^{OTL} aims at identifying outliers of newly collected data points, i.e., delay and topology measurements:

$$\begin{aligned} TC^{OTL} =& \langle E, L, O, N \rangle \\ =& \langle \langle detect_outliers \rangle, \\ & \langle \langle test_data \rangle \rangle, \\ & \langle assert_no_outliers \rangle, 1 \rangle \end{aligned}$$

TC^{OTL} consists of one procedure which aims at detecting outliers present in the newly collected probes. Detecting outliers in the probes is necessary for two reasons: First, we do not want to classify outliers because the result of such predictions may be erroneous. Second, we want to incrementally add new data points to the data collection which are used to update the classifier (see workflow for further details). To that end, various unsupervised outlier detection algorithms are available, e.g., based on K-nearest neighbor graph (e.g., [268]) or one-class (or single-class) SVM (e.g., [269]). Detected anomal data points are marked as outliers and are removed from the data set. Finally, an instance of TC^{OTL} passes (O) if no outliers are detected.

The fifth test case of CT^{LV} predicts the location of newly observed probes:

$$\begin{aligned} TC^{PRD} =& \langle E, L, O, N \rangle \\ =& \langle \langle classify_new_probes \rangle, \\ & \langle \langle test_data, trained_classifier, expected_location \rangle \rangle, \\ & \langle assert_correct_location \rangle, 1 \rangle. \end{aligned}$$

TC^{PRD} has one procedure (E) which classifies, i.e., predict the location of newly collected probes for a particular location. To that end, it takes as input (L) the newly collected probes having the data structure shown in Figure 5.3, a trained model of the classifier and the expected location of the probes. An instance of TC^{PRD} passes (O) if the predicted location matches the expected location.

Lastly, the sixth test case of CT^{LV} updates the classifier based on the newly collected probes:

$$\begin{aligned} TC^{UPD} =& \langle E, L, O, N \rangle \\ =& \langle \langle train_classifier \rangle, \\ & \langle \langle classification_algorithm, augmented_training_data \rangle \rangle, \\ & \langle assert_trainingError < \delta_{\varepsilon} \rangle, 1 \rangle. \end{aligned}$$

As we can see, this test case is almost identical to the initial training which is conducted by test case TC^{TRN}: The first difference is that TC^{UPD} takes the augmented data set as input and retrains a new classifier based on this data (L). Further, TC^{UPD} passes (O) if the training error observed *after updating* the classifier is lower than δ_{ε}.

[72]Note that cross-validation can be used to find the model that fits the training data best.

5.3.3.2 Test suites

The continuous test CT^{LV} consists of six test suites which are explained hereafter. The first test suite is responsible for the initial conduct of delay and topology measurements of a cloud service's components locations:

$$\begin{aligned} TS_{collect}^{\langle IPD_1, TCPD_1, \ldots, IPD_l, TCPD_l \rangle} =& \langle \mathcal{TC}, I, F, T \rangle \\ =& \langle \langle TC_1^{IPD}, TC_1^{TCPD} \ldots TC_l^{IPD}, TC_l^{TCPD} \rangle, 1000, 5, \langle 30 \rangle \rangle. \end{aligned}$$

The test suite binds all test cases which are available for any location l, i.e., $TC_1^{IPD}, TC_1^{TCPD} \ldots TC_l^{IPD}, TC_l^{TCPD}$. Execution of this test suite can be triggered 1000 times successively (I) where each run executes 30 seconds after the previous one completed (T), with an additional offset of 5 seconds (F).

The second test suite uses the collected data points to train a classifier:

$$\begin{aligned} TS_{train}^{\langle TRN \rangle} =& \langle \mathcal{TC}, I, F, T \rangle \\ =& \langle \langle TC^{TRN} \rangle, 1, 0, \langle 0 \rangle \rangle. \end{aligned}$$

This test suite binds the test case TC^{TRN} and is executed once (I). Its execution is triggered instantly, i.e., it has an offset (F) and interval (I) of 0.

The third test suite controls collection of probes for each location l:

$$\begin{aligned} TS_{probe}^{\langle IPD_1, TCPD_1, \ldots, IPD_l, TCPD_l \rangle} =& \langle \mathcal{TC}, I, F, T \rangle \\ =& \langle \langle TC_1^{IPD}, TC_1^{TCPD} \ldots TC_l^{IPD}, TC_l^{TCPD} \rangle, 10, 5, \langle 0 \rangle \rangle. \end{aligned}$$

Similar to $TS_{collect}$, this test suite binds all test cases which collect delay and topology measurements for any location l. It can execute ten successive times, without any waiting time (T) but with an offset (F) of five seconds. This means that TS_{probe} collects a batch of ten new probes for any location l.

The fourth test suite controls detection of outliers in the newly collected probes:

$$\begin{aligned} TS_{outlier}^{\langle OTL \rangle} =& \langle \mathcal{TC}, I, F, T \rangle \\ =& \langle \langle TC^{OTL} \rangle, 1, 0, \langle 0 \rangle \rangle. \end{aligned}$$

This test suite only binds the test case TC^{OTL}. It is executed once, without any waiting time (T) or offset (F).

The fifth test suite is responsible for controlling the prediction of the location of collected measurements:

$$\begin{aligned} TS_{predict}^{\langle PRD_1, PRD_2, \ldots, PRD_l \rangle} =& \langle \mathcal{TC}, I, F, T \rangle \\ =& \langle \langle TC_1^{PRD}, TC_2^{PRD}, \ldots TC_l^{PRD} \rangle, 1, 0, \langle 0 \rangle \rangle. \end{aligned}$$

This test suite binds all test cases which predict locations using delay and topology measurements for any location l. It only executes one single time, not having any waiting time (T) or offset (F).

Finally, the sixth test suite uses the augmented data set to update, i.e., retrain a new classifier:

$$\begin{aligned} TS_{update}^{\langle UPD \rangle} =& \langle \mathcal{TC}, I, F, T \rangle \\ =& \langle \langle TC^{UPD} \rangle, 1, 0, \langle 0 \rangle \rangle. \end{aligned}$$

This test suite binds the test case TC^{UPD} and is executed once (I). Its execution is triggered instantly, i.e., it has an offset (F) and interval (I) of 0.

5.3.3.3 Workflow

Figure 5.4 shows the workflow of the continuous test CT^{LV}: It starts with the test suite $TS_{collect}$ to collect the initial data points which are then used by the following test suite TS_{train} to train the initial classifier. Recall that the test suite TS_{train} only uses the test case TC^{TRN} which passes if the misclassification error observed during training (ε) is less than $\hat{\varepsilon}$. Thus, test suite TS_{train} only passes if training accuracy is sufficiently high, otherwise the continuous test CT^{LV} terminates because we assume that only having an initial classifier with a training error larger than $\hat{\varepsilon}$ will not lead to meaningful predictions. In the latter case, CT^{LV} terminates and has to be restarted from scratch. Possible adjustments to increase the performance of the initial classifier include increasing the number of delay and topology measurements per location, i.e., the iteration of test suite $TS_{collect}$, as well as selecting different parameter used for cross-validation, e.g., increasing 3-fold to 5-fold cross-validation.

If TS_{train} passes, TS_{probe} is the next test suite which is executed. It conducts new measurements of the cloud service's components presumed locations. These newly collected samples are used by the later following test suite $TS_{predict}$ to predict and validate their locations. To that end, test cases bound to TS_{probe} measure network latency and topology information in the same way as $TS_{collect}$ did for initial data collection. Thereafter, obtained delay and topology measurements are transformed into feature vector shown in Figure 5.3. If all probes on the IP and TCP Layer have been collected successfully, TS_{probe} passes, otherwise iterations of TS_{probe} restart.

If TS_{probe} passes, the test suite $TS_{outlier}$ is executed next. It inspects the newly collected data points provided by TS_{probe} and identifies as well as filters out outliers. Regardless of $TS_{outlier}$ passing or failing, the remaining samples are provided to the next test suite $TS_{predict}$ which predicts the location for the supplied samples and compares the predictions with the expected values. If the predictions for the newly supplied data points match the expected locations, then $TS_{predict}$ passes.

Independent of $TS_{predict}$ passing or failing, the next test suite to be executed is TS_{update} which uses the augmented data set to retrain the classifier. At this point, it is important to note that the increasing size of the data set may become too large for some classifier, e.g., for SVM. Therefore, we use a process parameter which allows controlling the upper bound of the training set size. If this bound is reached, then we apply a *sliding window* approach, that is, each time newly collected probes are added to the data set for retraining, the same amount of the oldest data points in the training set are discarded. Recall that the test suite TS_{update} only binds the test case TC^{UPD} which passes if the training error (ε) observed *after updating* is less than δ_{ε}. If TS_{update} fails, then CT^{LV} terminates and has to be restarted from scratch.

If test suite TS_{update} passes, the workflow now again triggers execution of TS_{probe} which collects new probes for each cloud service component whose location we aim to validate, continuing with $TS_{outlier}$, $TS_{predict}$ and so forth.

5.3.3.4 Test metric

The goal of the test CT^{LV} is to repeatedly validate the locations of cloud service components. To that end, we can compute a test metric which uses the results of test suite TS^{PRD} as input. At this point, it is crucial to note that naively treating results produced by TS^{PRD} to compute, e.g., the universal test metric Basic-Result-Counter (*brC*) can lead to erroneous conclusions about whether a cloud service's component is hosted at a valid location. The reason for this is as follows: Recall that during training of the initial classifier, we defined

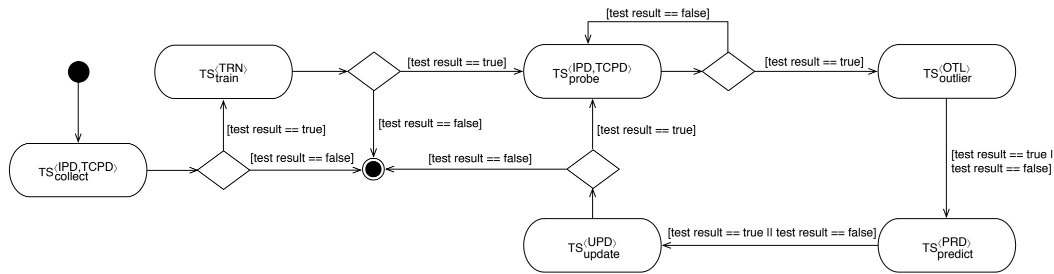


Figure 5.4: Workflow of continuously validating the locations of cloud service components (CT^{LV})

the upper bound for the training error $\hat{\varepsilon}$, i.e., the overall error we allow the classifier to make when predicting a location during training. Thus, we cannot exclude the possibility that a classifier's prediction which is made by test suite TS^{PRD} is erroneous. Such errors can lead to *incorrectly invalidating* (false negatives) a cloud service component's location. Consequently, we may not want to invalidate a component's location based on a single test result of TS^{PRD}.

Note that false positive classifications, that is, incorrectly validating a component's location are neglected at this point since those imply that the cloud service provider is actively cheating on the measurement: In this case, the expected location of a component matches the predicted one but the measured data point actually stems from a different location.

Invalidation window In order to minimize the probability to incorrectly invalidate the location a cloud service's components, an obvious approach is to consider a sequence of predictions for a location l within a particular time interval. Consider, for example, having observed a sequence of predictions for location l where the first three agree with the expected component's location (+) while the last two predictions indicate disagreement (−), i.e., $\langle +,+,+,-,- \rangle$. This raises the question whether this sequence indicates that location l is valid or not? A basic solution is a simple vote, that is, rely on highest frequency and thus, in this example, validate the location. This naive approach has many drawbacks since it, for example, requires to arbitrarily define an uneven number of successive predictions to be considered when voting.

In order to address the above issue, we propose to explicitly control the probability of incorrectly invalidating a location. To that end, we introduce the *invalidation window* w_l^- which is the sequence of successive predictions disagreeing with the expected location which is required to decide that a location is indeed invalid. The invalidation window can be obtained from the observed training error ε as follows: We assume that ε is independent, that is, any prediction indicating an invalid location has the probability ε to be incorrect. On this basis, we define the invalidation error $\upsilon_l^- \in [0,1]$ which describes the probability that the number of $|w_l^-|$ successive predictions for location l disagreeing with the expected location are incorrect:

$$\upsilon_l^- = \varepsilon_1 \times \varepsilon_2 \times \cdots \times \varepsilon_{|w_l^-|} = \prod_{i=1}^{|w_l^-|} \varepsilon_i = \varepsilon^{|w_l^-|}. \tag{5.1}$$

The key idea here is to use υ_l^- as a parameter which we configure to define the maximum probability of making an error when invalidating a location l. In order to formulate this notion, we reformulate and adapt equation 5.1 to create the following inequation:

$$|w_l^-| \geq \frac{\log(\upsilon_l^-)}{\log(\varepsilon)}. \tag{5.2}$$

We can use Inequation 5.2 to calculate the minimum number of successive predictions which are required to be observed in order to invalidate a location l for given υ_l^- and ε. Naturally, if we are willing to accept an invalidation error equal or higher than the training error, that is, $\upsilon_l^- \geq \varepsilon$, then the invalidation window size is $|w_l^-| = 1$.

Consider, as an example, that we are willing to accept a maximum probability of making an error when invalidating location l of 0.0001%, i.e., $\upsilon_l = 0.000001$. Let's further assume that the observed training error of the classifier is 2%, i.e., $\varepsilon = 0.02$. The minimum number of successive predictions to observe indicating disagreement is

$$|w_l^-| \geq \frac{\log(0.000001)}{\log(0.02)} \approx 3.53.$$

Thus, in this example, the minimum number of successive predictions disagreeing with the expected location which are needed to conclude that a location l of cloud service component is invalid – i.e., not hosted at its presumed location – is four.

Note that there is a subtle detail to this definition of the invalidation window size: The observed training error ε is calculated based on *all* incorrectly classified locations observed when applying the classifier to the training set while the invalidation error υ_l^- is defined for an *individual* location l. From the perspective of the classifier's performance, the formulation of Inequation 5.2 can thus be interpreted as the worst case expectation: Incorrectly classified samples of a single location l may be responsible for the entire observed training error ε while any other location contained in the training set actually performing perfect is assumed to possess the same error.

Finally, the invalidation window size is adjusted after each successful completion of test suite TS_{update} using the current training error observed after updating. Therefore, the invalidation window size dynamically adapts to the varying performance of the classifier observed during retraining.

To summarize: Since a result produced by TS^{PRD} is expected to be erroneous with probability ε, we consider a sequence of successive negative test results to invalidate a location which we refer to as invalidation window w_l^-. The size of w_l^- is derived from the invalidation error υ_l^-, i.e the maximum probability of incorrectly invalidating a location we are willing to tolerate, and the observed training error ε. The invalidation window size dynamically adapts according to the training error observed after updating the classifier.

Configuring the invalidation window Applying the concept of the invalidation window to the test metric brC requires the following final considerations: Each time a prediction made by TS^{PRD} for a location l disagrees with the expected location, this constitutes a negative test result for that location which is added to the invalidation window w_l^-. Assuming, for example, w_l^- has a size of three, then observing three negative test results produced by TS^{PRD} for location l completes the invalidation window, that is, the test metric produces a negative test result $br^N_{\upsilon_l^-}$. Any $br^N_{\upsilon_l^-}$ observed during the continuous test CT^{LV} is counted using $brC^N_{\upsilon_l^-}$. If TS^{PRD} only indicates that two predictions do not agree with expectation for l, then these two negative test results are ignored because we assume that they are prediction errors.

The above paragraph leads to the question what maximum invalidation window size is required? It is obvious that the upper bound for the invalidation window size is the number of predictions supplied by test suite $TS_{predict}$: Consider, as an example, each execution of $TS_{predict}$ makes ten new predictions for a location l. Consequently, the maximum number of predictions which may incorrectly indicate disagreement with the expected location is also ten.

This, in turn, implies that ten newly collected probes for a location l are provided to $TS_{predict}$ by ten iterations of test suite TS_{probe}. Therefore, neglecting potentially filtered outliers by $TS_{outlier}$, the maximum invalidation window size directly depends on the iterations of TS_{probe}.

In order to configure the maximum invalidation window size – and thus define minimum number of iterations of TS_{probe} – in a non-arbitrary way, we can draw on the notion formulated by Equation 5.1 which relates the invalidation error to the training error and the invalidation window size. The idea here is to derive the maximum invalidation window size based on the maximum training error δ_ε which is observed after having retrained on the augmented data set during test suite TS_{update}. Recall that TS_{update} has only one test case TC^{UPD} which updates the current classifier and uses the manually configured performance constraint δ_ε as test oracle.

Consider the following example of obtaining the maximum invalidation window size: Assuming that the desired invalidation error for a location l is 0.003%, that is, $\upsilon_l^- = 0.00003$, and that we are willing to tolerate a maximum training error observed after updating of $\delta_\varepsilon = 20\%$. Through adapting Equation 5.1, we can determine the maximum invalidation window size as follows:

$$|\tilde{w}_l^-| \leq \frac{\log(\upsilon_l^-)}{\log(\delta_\varepsilon)} \leq \frac{\log(0.00003)}{\log(0.2)} \leq 6.47. \tag{5.3}$$

Thus the maximum invalidation window size is six which means that iterations of TS_{probe} have to be configured to be (at least) six. A maximum invalidation window size of 6, in turn, corresponds to a maximum training error observed after updating of

$$\delta_\varepsilon = 0.00003^{\left(\frac{1}{6}\right)} \approx 0.1763.$$

To summarize: Configuring the number of iteration of TS_{probe} to $I = 6$ implies that the invalidation window size $|w_l^-|$ can grow to a maximum size of six. An invalidation window size larger than six, in turn, implies that the observed training error after updating (δ_ε) may exceed the tolerance of 20%.

Finally, our test metric bears the risk of taking a prediction error for what is actually a cloud service component hosted at an invalid location. In our above example, the two failed instances of TS^{PRD} may not be caused by an error of the classifier but may actually result from an invalid location. To counter this misinterpretation of results produced by TS^{PRD}, two parameters are crucial: First, the time between successive probes taken by TS_{probe} has to be chosen suitably short such that relocating the cloud service component while CT^{LV} executes TS_{probe} becomes rather unlikely. Second, the smaller we define the error of incorrectly invalidating a cloud service component's location, the more successive failed test results produced by TS^{PRD} are needed to invalidate a location. Thus, the desired size of validation error υ_l should be configured carefully.

5.3.4 Implementation and experiment

In this section, we describe implementation as well as experimental evaluation of our approach to test the locations of cloud service components.

5.3.4.1 Environment and setup

Hereafter, the main components of the experiment are outlined.

Cloud service components under test Figure 5.5 shows the distribution of AWS global infrastructure: In AWS terminology, there are 16 geographical *regions* (orange circles) each of which having multiple *availability zones* (indicated by the number in the orange circles) as well as planned regions (green circles)[73]. These regions include [3]: Oregon, Northern California, Northern Virginia, Ohio, Central Canada, Sao Paulo, Ireland, Frankfurt, London, Singapore, Sydney, Tokyo, Seoul, Mumbai, Beijing, as well as AWS Gov Cloud[74].

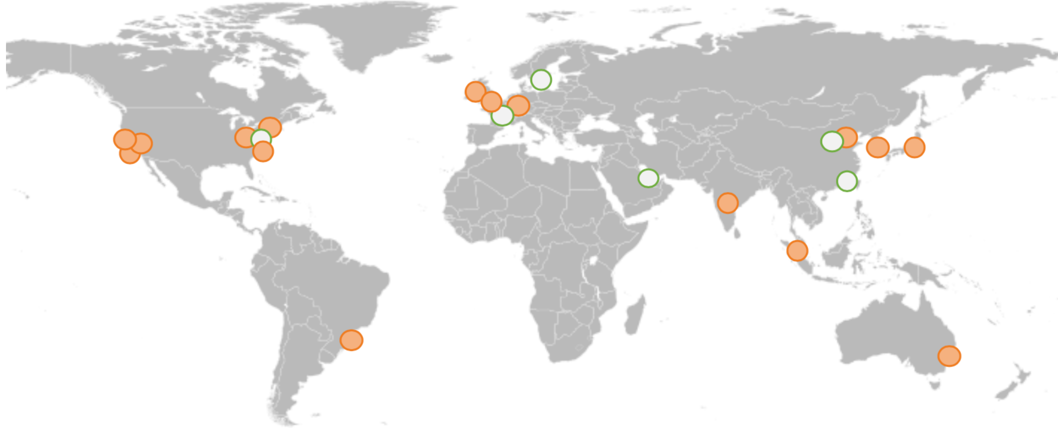


Figure 5.5: AWS Global Infrastructure (Figure based on [3])

Within our experiment, we deployed 14 Amazon EC2 instances, i.e., IaaS instances in total, each hosted in one of the above regions, excluding Beijing[75] and AWS Gov Cloud. Each instance had identical configurations, that is, running Ubuntu Server 16.04 LTS with 1 vCPU, 0.5 GB main storage and 8 GB persist storage. Further, each instance was associated with a publicly reachable IP address and its security group is configured to allow for traffic via ICMP, via TCP on port 22 and via SSH from and to the landmark. Lastly, the landmark from which delay and topology measurements were conducted was also hosted on AWS, at region Frankfurt.

Implementation of test cases Delay and topology measurements are conducted by the test cases TC^{IPD} and TC^{TCPD}: The first one uses *MTR*[76], a common network diagnostic tool which allows determining if a remote host is reachable over an IP network as well as to identify the path packets took to the remote host. The second one uses *Nping*[77] which offers the possibility to measure response times on the TCP Layer. Recall that test suites $TS_{collect}^{\langle IPD,TCPD \rangle}$ as well as $TS_{probe}^{\langle IPD,TCPD \rangle}$ (see Section 5.3.3.2) always trigger execution of TC^{IPD} and TC^{TCPD} per target location. Once both TC^{IPD} and TC^{TCPD} have successfully completed, their output is parsed and a data point following the data structure shown in Figure 5.3 is stored. Finally, it is important to note that – as already mentioned in the previous

[73] These planned regions include Paris, Stockholm and Ningxia. For further information see [3].

[74] AWS GovCloud allows US government agencies to use cloud resources in compliance with US-specific regulatory requirements. According to AWS, the GovCloud is located in the Northwestern region of the United States. For further information see `https://aws.amazon.com/govcloud-us/faqs/` [Accessed: 2018-12-13]

[75] As of the time when conducting the experiment to continuously validate cloud service component's location, the AWS region Beijing was not available.

[76] `https://linux.die.net/man/8/mtr` [Accessed: 2018-12-13]

[77] `https://nmap.org/nping/` [Accessed: 2018-12-13]

section – the delay and topology measurements are all conducted from an EC2 instance hosted in the region Frankfurt.

Training and updating the classifier is the responsibility of test case TC^{TRN} and TC^{UPD}: Both test case instances use the same implementation based on *scikit-learn*[78] to (re-)train a classifier based on *Linear Support Vector Classification (LinearSVC)*[79] with 5-fold cross-validation. This classifier takes as input the measurements provided by the test cases TC^{IPD} and TC^{TCPD} where each class label is given by the target location for which the measurements have been conducted. The complete set of parameter passed to train the instance of LinearSVC can be found in Appendix A.

The test case TC^{OTL} aims at detecting outliers in newly collected probes, that is, data points provided by the test cases TC^{IPD} and TC^{TCPD} for each target location bound to the test suite $TS^{IPD,TCPD}_{probe}$. To that end, we also use *scikit-learn* to conduct unsupervised outlier detection using *OneClassSVM*[80]. The complete set of parameter passed to the instance of OneClassSVM can be found in Appendix A.

Finally, the test case TC^{PRD} predicts class labels of newly collected data points which are provided by the test case TC^{IPD} and TC^{TCPD} (bound to test suite $TS^{IPD,TCPD}_{probe}$). We use *scikit-learn* and the LinearSVC which we trained beforehand to predict the location of an EC2 instance.

5.3.4.2 Experiment results

Hereafter, we present the experimental results obtained when executing CT^{LV}.

Data set For the 14 EC2 instances described above, we collect a total of 139699 data points. The collection is split up into two periods, the first started at 17th December 2016, 12:04:40 (UTC) and ended at December 23rd 2016, 10:29:52 (UTC) while the second period started at 25th December 2016, 12:20:27 (UTC) and ended at 3rd January 2017, 10:34:46 (UTC). Each single data point contains the information described by the data structure specification shown in Figure 5.3.

Experiment and evaluation In order to initialize CT^{LV}, we define an upper bound for the misclassification error observed during training of the initial classifier of $\hat{\varepsilon} = 3.27\%$ which is achieved by initially collecting 13979 records or using the first ≈10% of the data set to train the initial classifier.

Furthermore, for each of the 14 locations, we chose the identical invalidation error of $\upsilon_l^- = 0.001\%$ as an example value. Recall that υ_l^- defines the maximum error we are willing to tolerate when incorrectly invalidating a location. Also, we are willing to tolerate a maximum training error observed after updating the classifier of $\delta_\varepsilon = 35\%$. The resulting maximum invalidation windows size therefore is (following Equation 5.3)

$$|\tilde{w}_l^-| \leq \frac{\log(\upsilon_l^-)}{\log(\delta_\varepsilon)} \leq \frac{\log(0.00001)}{\log(0.35)} \leq 10.97.$$

[78] `http://scikit-learn.org/`
[Accessed: 2018-12-13]

[79] `http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html`
[Accessed: 2018-12-13]

[80] `http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html`
[Accessed: 2018-12-13]

A maximum invalidation window size of ten means that iterations of TS_{probe} have to be (at least) $I = 10$ and corresponds to maximum tolerated training error observed after updating of

$$\delta_{\varepsilon} = 0.00001^{\left(\frac{1}{10}\right)} \approx 0.3162.$$

Applying the results described in the previous paragraph, we split the remaining 90% or 125720 up into 898 successive batches where each batch contains ten probes per location, that is, 140 probes per batch. Starting with the oldest one, each batch *simulates* the execution of test suite $TS_{probe}^{\langle IDP,TCPD\rangle}$ by supplying 140 newly collected probes to test suite $TS_{outlier}^{\langle OTL\rangle}$, thereby triggering the workflow shown in Figure 5.4 of Section 5.3.3.3: Having completed outlier detection, locations of all non-outliers are predicted and the predictions are compared with the expected locations to compute the proportion of the correctly classified samples per batch, i.e., the test accuracy per batch (Figure 5.6). Thereafter, the initial data set is augmented and the classifier is updated, that is, retrained. Since there are 898 batches each of which supplying 140 newly collected probes, we update the classifier during our experiment for 898 times. After each update, we observe the training error ε. Figure 5.8 shows how the training error evolves over time while Figure 5.7 shows the number of outliers filtered per batch.

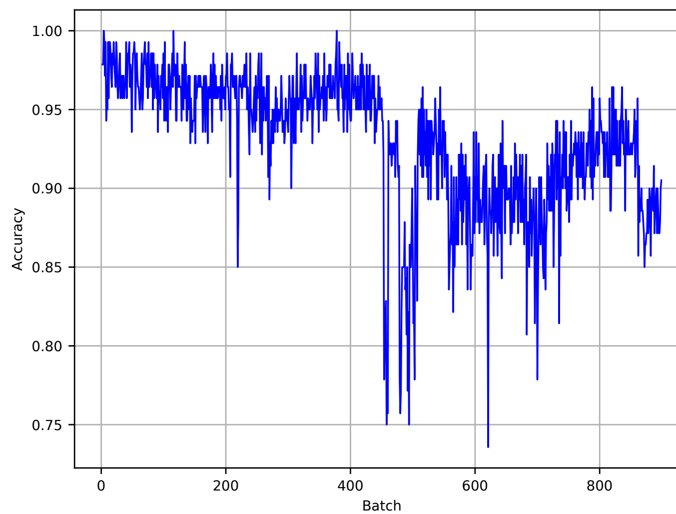


Figure 5.6: Evolving test accuracy per batch over time of CT^{LV}

Since we use the LinearSVC, the required storage and compute resources increase quickly with increasing size of the training set[81]. As a result, we cannot increase the size of the training set arbitrarily but have to define an upper bound, in our case 30000 data points. Once our process reaches this upper bound, we use a sliding window to augment the training set, i.e., for each newly collected batch of 140 probes (minus those filtered by outlier detection), we remove the same amount of the oldest data points in the training set.

For each batch, we observe the correctly classified locations per batch, that is, the *test accuracy per batch*, as well as the correctly classified locations during training, i.e., *training*

[81] http://scikit-learn.org/stable/modules/svm.html#complexity [Accessed: 2018-12-13]

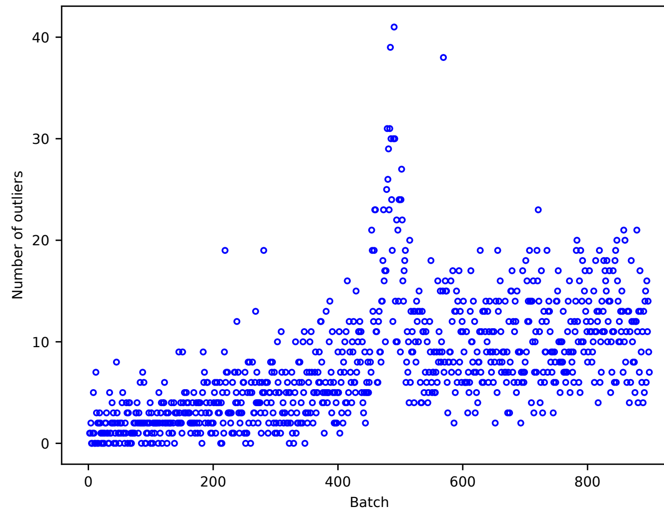


Figure 5.7: Filtered outliers per batch of CT^{LV}

accuracy. We make use of these statistics to summarize the outcome of the experiment (Table 5.2) by computing mean ($\bar{x}$), median ($\hat{x}$) standard deviation sd, min, and max. Additionally, Figure 5.6 shows how the test accuracy evolves over time, that is, with increasing number of batches consumed by the workflow of CT^{LV}. It is important to note at this point that we assume that all 14 instance reside at their claimed location at the time of the experiment, that is, the observed deviations from expected locations result from errors the classifier makes.

The experimental results can be summarized as follows (see Table 5.2): The mean test accuracy per batch is 92.96% which translates to a mean misclassification rate of 7.04% per batch. At batch number 621, the accuracy drops below 75%, with a minimum at 73.57%. Further, the mean training accuracy is 94.13% with a minimum of 90.07%.

Furthermore, through inspection of how the training error shown in Figure 5.8 evolves over time, a sharp increase at batch 455 can be observed. In this context, Figure 5.9 illustrates how the invalidation windows size compensates for this increase in training error: After having consumed batch number 455, the increased training error after updating the classifier is used to adapt the invalidation window size from $|w_l^-| = 4$ to $|w_l^-| = 5$.

It is important to note that the invalidation window size adaption observed during the course of the experiment ranged from three to five. However, the invalidation window was *never* completed for any location for any batch, i.e., the test metric never produced a negative test result $br_{v_l^-}^N$ invalidating a location. Put differently: Since we are assuming that all 14 instances are hosted at their claimed location during the experiment, we can conclude that no location of the cloud service components under validation was incorrectly invalidated. This is, after all, the expected and desired result of this experiment because it shows that the design of the continuous test CT^{LV} is capable of compensating prediction errors a classifier may make, thus avoiding to incorrectly invalidate a location of a cloud service component.

Table 5.2: Results of continuous location validation of 14 AWS EC2 instances using 10% of total data set as initial training data and 898 successive batches with each 140 newly collected probes

Parameter	$\bar{x}$ (%)	$\hat{x}$ (%)	*sd* (%)	*max* (%)	*min* (%)
Test accuracy per batch	92.96	94.28	4.35	100	73.57
Training accuracy	94.13	95.41	2.47	97.91	90.07

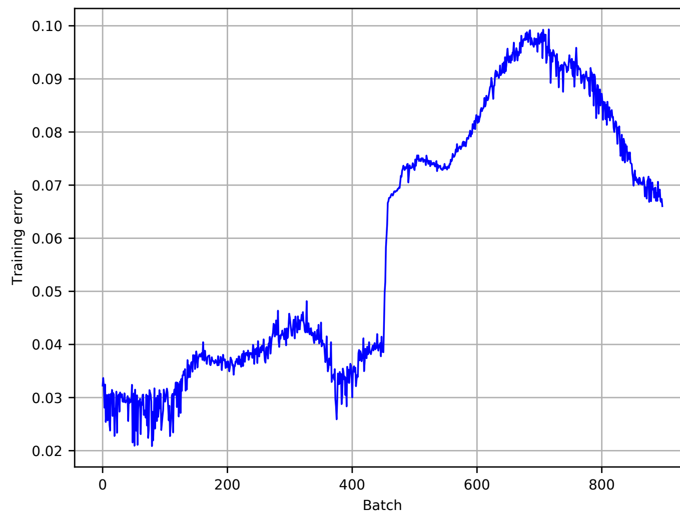


Figure 5.8: Evolving training error ε observed per batch over time of CT^{LV}

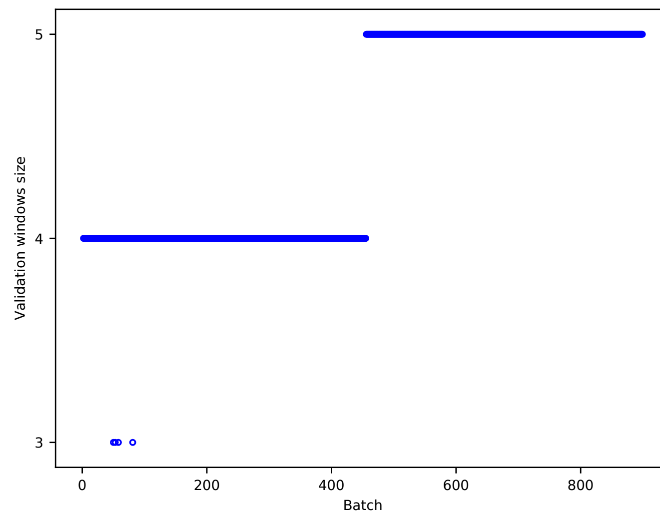


Figure 5.9: Adaption of invalidation window size per batch over time of CT^{LV}

5.4 Continuously testing secure communication configuration

In this test scenario, we continuously test whether the configuration of a cloud service allows to securely communicate with its endpoints. We begin with defining what the property *secure communication configuration* means in context of cloud services and outline the test scenario. Thereafter, we describe the general characteristics of the test scenario (Section 5.4.2) and describe the test design (Section 5.4.3). Lastly, we present implementation as well as experimental results of using our test to check the secure communication configuration of SaaS instances which are provided on top of IaaS delivered by OpenStack (Section 5.4.4). Parts of this section have been published in [127].

5.4.1 Property description and overview of test scenario

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 customers 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 [270]. 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 [271].

We propose the test CT^{SC} which continuously checks whether we can securely communicate with the endpoint of a cloud service. For this purpose, we continuously evaluate if the SSL/TLS configuration of a cloud service's web server allows to securely communicate with the service.

The workings of the test can be summarized as follows (a detailed description of the test will be provided in Section 5.4.3): Every ten seconds, CT^{SC} first executes a precondition test where it establishes that the endpoint of the cloud services is reachable via ICMP and TCP. In case these preconditions are satisfied, following the precondition test, CT^{SC} then tests the SSL/TLS configuration of the endpoint. 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, we are using *Precondition as specialized test suites* introduced in Section 5.4.3.2. This means that if one or both precondition test cases fail, then testing of the SSL/TLS configuration is *not* executed. In this case, precondition are tested again in the following iteration, i.e., ten seconds after the previous precondition tests have completed. Finally, the results produced by CT^{SC} will tell us how often the SSL/TLS configuration of the cloud service under test is insecure and how long it takes to fix these misconfigurations.

5.4.2 General characteristics of the test scenario

This section describes the general characteristics of the scenario to continuously test the secure communication configuration of cloud service components.

5.4.2.1 Supported cloud-specific controls

The control *KRY-02 Encryption of data for transmission (transport encryption)* of BSI C5 [31] demands that

> *"Procedures and technical safeguards for strong encryption and authentication for the transmission of data of the cloud customers (e. g. electronic messages transported via public networks) are established."*.

The continuous test CT^{SC} can support certification of a cloud service according to this control because it analyses if a cloud service's web server supports weak cipher suites, possesses known vulnerabilities or uses self-signed certificates. Furthermore, in [253] the BSI maps the control *KRY-02* to the control *EKM-03: Encryption & Key Management Sensitive Data Protection* of CSA's Cloud Control Matrix (CCM) [22] which requires that

> *"Policies and procedures shall be established, and supporting business processes and technical measures implemented, for the use of encryption protocols for protection of sensitive data in storage (e.g., file servers, databases, and end-user workstations), data in use (memory), and data in transmission (e.g., system interfaces, over public networks, and electronic messaging) as per applicable legal, statutory, and regulatory compliance obligations."*.

This control is less specific than *KRY-02* of BSI C5 as it broadly requires implementing technical measures to protect sensitive data in transmission over public network. Thus, CT^{SC} can also be used to produce evidence to support the certification of a cloud service according to this control.

Finally, both BSI and the CCM map *KRY-02* and *EKM-03* to the control *A.14.1.2 Securing application services on public networks* of ISO/IEC 27001:2013. Thus, CT^{SC} can also be employed to support the certification of a cloud service according to *A.14.1.2*.

5.4.2.2 Supported cloud service models

The test CT^{SC} repeatedly checks whether the SSL/TLS configuration of a web server of a cloud service possesses some known security vulnerability, e.g., OpenSSL Heartbleed Bug [272], uses self-signed certificates, or supports weak cipher suites.

As already mentioned in the introduction, SaaS applications use HTTPS for secure communication between a cloud service customer who accesses the SaaS application via browsers. Thus, we can use the hostname which points to the website of the SaaS application to conduct our test. Similarly, a PaaS application such as a database service may expose a RESTful API to, e.g., allow for administration and querying. Using TLS to secure data in transit to and from a RESTful APIs is the best practice [273]. Thus, we can use the hostname of a RESTful API's base URL to continuously check whether the web server of the PaaS application is securely configured.

Usually SSH is used to securely communicate with virtual machines, i.e., IaaS provided by a remote host. Since CT^{SC} focuses on testing the security of SSL/TLS configurations, it cannot be applied to IaaS.

5.4.2.3 Implementation independence

Nginx[82], Apache HTTP Server[83] and IIS[84] are today's dominant web server implementations[85] which SaaS applications may use. All of them support TLS and thus the test CT^{SC} can be used to repeatedly check whether their configurations are secure.

5.4.2.4 Minimally invasive integration

If the firewall rules of the cloud service under test allow communication with its endpoints, that is, if the web server component of a SaaS or PaaS application can be reached from the host where the continuous test CT^{SC} is deployed, then no change of the composition or configuration of the cloud service is needed. While we can expect this to be the case for most SaaS applications, access to endpoints of a PaaS application such as a database service may be restricted to some IP addresses. Thus, in the latter case, firewall rules will have to be changed accordingly in order for CT^{SC} to function properly and deliver meaningful test results.

5.4.3 Test design

In this section, we describe the test CT^{SC} according to the framework introduced in the Chapter 4. Note that values used to configure CT^{SC} (e.g., length of test suite intervals) are to be understood as examples. These values may differ depending on the test's use case and the interpretation of the controls it aims to provide evidence for.

5.4.3.1 Test case

The continuous test CT^{SC} consists of the following single test case:

$$
\begin{aligned}
TC^{SSL} =& \langle E, L, O, N \rangle \\
=& \langle \mathit{analyze_SSL_TLS_configuration} \rangle, \\
=& \langle \langle \mathit{host, port} \rangle \rangle, \\
=& \langle \mathit{assert_no_known_SSL_TLS_vulnerability, assert_no_blacklisted_cipher_suites,} \\
& \mathit{assert_SCSV_support, assert_no_self_signed_certificate} \rangle, 1 \rangle.
\end{aligned}
$$

The test case TC^{SSL} has a single procedure (E) to analyze the SSL/TLS configuration of a web server. To that end, the test case takes as input parameters (L) the IP address or the hostname of the cloud service's web server as well as its port. In order to evaluate the results produced by the test case (O), we validate that the web server does not possess a known SSL/TLS vulnerability including the OpenSSL Heartbleed Bug [272], CRIME [274] and OpenSSL CCS Injection [275]. Furthermore, the test case results must not indicate a web server supporting blacklisted cipher suites. Also, the web server has to support *TLS fallback signaling cipher suite value (scsv)* which aims at preventing protocol downgrading attacks [276] as well as secure session renegotiation [277]. Finally, a web server must not allow

[82] `https://nginx.org/` [Accessed: 2018-12-13]

[83] `https://httpd.apache.org/` [Accessed: 2018-12-13]

[84] `https://www.iis.net/` [Accessed: 2018-12-13]

[85] `https://news.netcraft.com/archives/2017/01/12/january-2017-web-server-survey.html` [Accessed: 2018-12-13]

self-signed certificates. If all of these assertions hold, then an instance of TC^{SSL} passes. Lastly, the ordering number of the test case is 1.

5.4.3.2 Preconditions

The test CT^{SC} may fail not because of, e.g., an insecure cipher suite but because the web server of the cloud service cannot be reached at all. In order to avoid test suite run results incorrectly indicating (false negative test results) that the configuration of a cloud service's web server is insecure, we define the following two preconditions:

$$\begin{aligned} TC^{\widehat{ICMP}} =& \langle E, L, O, N \rangle \\ =& \langle \langle \textit{measure_delta_ICMP_Echo_Request_to_ICMP_Time_Reply_packets} \rangle, \\ & \langle \langle \textit{host, packet_count} \rangle \rangle, \\ & \langle \textit{assert_rtt_avg} < \$^{ICMP}_{avg}, \textit{assert_rtt_sd} < \$^{ICMP}_{sd} \rangle, 1 \rangle \end{aligned}$$

and

$$\begin{aligned} TC^{\widehat{TCP}} =& \langle E, L, O, N \rangle \\ =& \langle \langle \textit{measure_delta_SYN_to_SYN-ACK_TCP_segment} \rangle, \\ & \langle \langle \textit{host, probe_count, port} \rangle \rangle, \\ & \langle \textit{assert_average_response_time} < \$^{TCP}_{avg}, \textit{assert_max_response_time} < \$^{TCP}_{max} \rangle, 1 \rangle. \end{aligned}$$

As described in Section 4.3.6, preconditions are a type of test case which aims to determine whether the assumptions about the test environment are satisfied. In the above case, we define two precondition test cases which tests if we can reach the target host on the Internet Layer and on the Transport Layer. Note that the definition of both precondition test cases are identical to the test cases we defined for the test of availability CT^{AV} (Section 5.2). This illustrates how test case definitions can be efficiently reused, in this case to test preconditions.

5.4.3.3 Test suites

The continuous test CT^{SC} defines the following two test suites:

$$\begin{aligned} TS^{\langle \widehat{ICMP}, \widehat{TCP} \rangle} =& \langle \mathcal{T}C, I, F, T \rangle \\ =& \langle \langle TC^{\widehat{ICMP}}, TC^{\widehat{TCP}} \rangle, +\infty, 0, \langle 10 \rangle \rangle \end{aligned}$$

and

$$\begin{aligned} TS^{\langle SSL \rangle} =& \langle \mathcal{T}C, I, F, T \rangle \\ =& \langle \langle TC^{SSL} \rangle, +\infty, 0, \langle 0 \rangle \rangle. \end{aligned}$$

As we can see, the precondition test cases $TC^{\widehat{ICMP}}$ and $TC^{\widehat{TCP}}$ are bound to the first test suite. This indicates that we are using the option *Precondition as specialized test suites* of our framework which was introduced in Section 4.3.6. A specialized test suite's execution is triggered (T) every 10 seconds after the previous one completed and the number of possible successive executions is set to infinity, i.e., $I = \langle +\infty \rangle$. The following paragraph on the workflow of CT^{SC} will explain how we use this specialized test suite to test preconditions.

The second test suite binds the main test case $TS^{\langle SSL \rangle}$. It does not possess any waiting time between successive test suite runs, that is, execution of this test suite is always triggered instantaneously (i.e., $F = 0$ and $T = 0$). Further, it does not have a maximum number of iteration but can be executed successively for an infinite number of times (i.e., $I = +\infty$).

5.4.3.4 Workflow

As described in Section 4.3.6, there is a base execution strategy which can be used with specialized test suites that test preconditions: As shown in Figure 5.10, the specialized test suite $TS^{\langle \widehat{ICMP}, \widehat{TCP} \rangle}$ is executed prior to the main test suite $TS^{\langle SSL \rangle}$. Thus, the workflow of CT^{SC} only executes the main test suite if the preconditions passed. If the specialized test suite fails, it is executed again after waiting 30 seconds. If it passes, the execution of the main test suite to check the SSL/TLS configuration is triggered immediately after the successful specialized test suite completed. This illustrates how preconditions can be used to control the workflow of a continuous test, allowing selecting and executing test suites according to environmental conditions discovered at runtime.

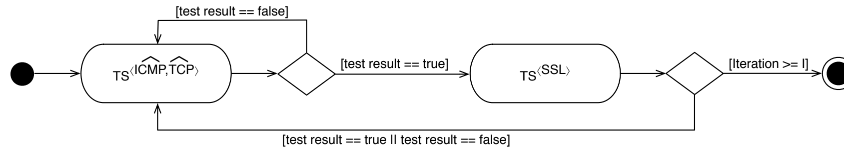


Figure 5.10: Workflow of continuously testing secure communication configuration (CT^{SC})

5.4.3.5 Test metric

The test CT^{SC} aims to obtain two types of measurement results: First, we want to know how often a cloud service's web server is not configured to securely communicate. To that end, we use the universal test metric $fpsC$ which counts the occurrences of Failed-Passed-Sequences (fps) (For a detailed explanation of fps see Section 4.3.5). Within our test scenario, a single fps is the period of time where the main test suite $TS^{\langle SSL \rangle}$ failed at least once and passed some time later, that is, the analysis of the SSL/TLS configuration identified some insecure configuration which was fixed thereafter.

Furthermore, the goal of CT^{SC} is to determine how long it takes to fix an incorrectly configured cloud service's web server. For this purpose, we can use the universal test metric $fpsD$ which tells us the duration of a fps.

Finally, since we are using the option *preconditions as specialized test suites* of our framework, we have to neglect the result of the specialized test suites during metric computation. This means that in order to calculate test metrics $fpsC$ and $fpsD$, we only consider test results which are produced by the main test suite $TS^{\langle SSL \rangle}$ and ignore any results of the specialized test suite $TS^{\langle \widehat{ICMP}, \widehat{TCP} \rangle}$.

5.4.4 Implementation and experiment

This section describes implementation as well as experimental evaluation of our approach to continuously test the secure communication configuration of cloud service components.

5.4.4.1 Environment and setup

Hereafter, the main components of the experiment are outlined.

Cloud service component under test As an example for a cloud service component under test, we choose an Apache HTTP Server which is deployed on an instance of OpenStack Mitaka with 2 vCPUs and 4 GB RAM and 40 GB volume (disk) running Ubuntu 15.10 Server. The instance is associated with a publicly reachable IP address and its security groups are configured to allow for traffic via ICMP, via TCP on port 22 and via SSH from and to the host where the test is deployed (see also paragraph *Deployment of continuous test* below). As already pointed out in Section 5.4.2.2, it depends on the service provided through this web server whether this cloud service component is used as part of a service of type PaaS or SaaS.

Test case implementation We use *sslyze*[86] to implement test case TC^{SSL}, a tool which connects to a web server and analyzes its TLS configuration. It supports detection of self-signed certificates, weak cipher suites, and also identifies known vulnerabilities such as OpenSSL Heartbleed Bug, insecure session renegotiation, and CRIME.

Sslyze is used by providing the hostname of the endpoint whose TLS configuration should be analyzed and the application specific port, e.g., *www.google.com:443*. Once the analysis is completed, sslyze returns the results, e.g., using a XML format. Usually, these results are then inspected by a human to determine whether the supported TLS configuration is secure. However, the test case TC^{SSL} parses the results of sslyze and checks whether it satisfies the specification provided in Section 5.4.3.1. If so, the test cases passes, otherwise it fails.

Implementation of preconditions In order to implement the preconditions $TC^{\widehat{ICMP}}$ and $TC^{\widehat{TCP}}$, we reuse the implementation of the test case TC^{ICMP} and TC^{TCP} of the test CT^{AV} which are described in Section 5.2.4.1. The specialized test suite $TS^{\langle\widehat{ICMP},\widehat{TCP}\rangle}$ triggers execution of both precondition tests concurrently. Once both precondition test cases have completed, the output of each tool, that is, Ping and Nping within $TC^{\widehat{ICMP}}$ and $TC^{\widehat{TCP}}$, respectively, is evaluated. To that end, we have to specify the assertions for each precondition:

- $\$^{ICMP}_{avg}$ = 100ms and $\$^{ICMP}_{sd}$ = 75ms, as well as
- $\$^{TCP}_{avg}$ = 75ms and $\$^{TCP}_{max}$ = 100ms.

Deployment of continuous test We deploy the continuous test CT^{SC} on a host different to the one where the cloud service component under test is running. Further, the host where CT^{SC} is running is attached to a different network than the host of the cloud service component under test. Yet both hosts reside in the identical building which is the reason why delays on the IP and TCP Layer measured previous to the experiment are relatively low, i.e., usually below 3 ms.

Inducing vulnerable TLS configurations We trigger temporary insecure communication configurations of our cloud service component under test by editing the configuration file of the Apache HTTP server such that it supports TLS communication using the weak

[86] `https://github.com/nabla-c0d3/sslyze` [Accessed: 2018-12-13]

cipher suite TLS_RSA_WITH_DES_CBC_SHA. We refer to the time during which the TLS configuration of the cloud service under test is insecure as a *vulnerability event*. Overall, we trigger 998 sequential events where TLS configuration is vulnerable. Each vulnerability event last at least 180 seconds plus randomly adding [0,60] seconds.

The accumulated duration of all vulnerability events observed during the experiment is 208867.41 seconds (≈ 58 hours 1 minute 7 seconds). Moreover, the mean duration of each vulnerability events was 209.29 seconds with a standard deviation of 17.43 seconds.

5.4.4.2 Experimental results

The entire experiment took ≈109.3 hours, i.e., ranging from the start of the first test event until the end of the last test. Table 5.3 shows the test statistics and test results produced by the test CT^{SC}: In total, CT^{SC} triggered 19514 tests of the TLS configuration of our cloud service component under test. When also including the precondition tests – recall that we are using *preconditions as a specialized test suite* – the total number of executed tests amounts to 39021. Producing evidence, i.e., executing a test of the TLS configuration plus evaluating the returned result took on average ≈5 seconds with a standard deviation of ≈4 seconds.

Out of the 998 vulnerable TLS configurations, CT^{SC} correctly detected 959 or 96.09% ($fpsC$). Furthermore, each $fpsD$ on average estimates that the duration of a correctly detected vulnerability event is 208.34 seconds ($fpsD$). Since a vulnerable TLS event lasted 209.29 seconds on average, we can conclude that CT^{SC} underestimates a vulnerability event on average by ≈1 second.

Table 5.3: Test statistics and results of continuous test CT^{SC}

Test statistics	number of tests	19514 (39021)
	mean duration test (sec)	5.02
	sd duration test (sec)	4.06
	min duration test (sec)	0.1
	max duration test (sec)	18.43
fpsC	correctly detected events	959
	correctly detected events (%)	96.09
fpsD	mean duration detected events (sec)	208.34
	sd duration detected events (sec)	19.64
	min duration of detected events (sec)	40.4
	max duration of detected events (sec)	244.33

5.5 Continuously testing secure interface configuration

In this test scenario, we continuously test whether cloud services components possesses a secure interface configuration. We begin with defining what the property *secure interface configuration* means in context of cloud services and outline the test scenario. Then we investigate the general characteristics of the test scenario (Section 5.5.2) and describe the test design (Section 5.5.3). Lastly, we present implementation as well as experimental results

of using our test to check the secure interface configuration of PaaS instances providing a Database-as-a-Service interface on top of IaaS delivered by OpenStack (Section 5.5.4). Parts of this section have been published in [127].

5.5.1 Property description and overview of test scenario

Secure interface configuration is another security properties of a cloud service which is satisfied if a cloud service 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 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)[87], 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*[88]. 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.

In order to determine whether a cloud service temporarily exposes interface due to insecure configurations, we propose the test CT^{SI}. CT^{SI} continuously probes the endpoints of a cloud service, either by hostname or IP address, for open ports which should not be publicly accessible.

The workings of this test can be summarized as follows (a detailed description of the test will be provided in Section 5.5.3): CT^{SI} tests every 30 seconds if the endpoint of the cloud service under test can be reached via ICMP 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 whose execution is triggered concurrently with the test for open ports. Thus, CT^{SI} makes use of the *preconditions as part of main test suites* option of our framework described in Section 4.3.6. 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 CT^{SI} show how long it takes the cloud service provider to fix insecure interface configurations of the cloud service under test.

5.5.2 General characteristics of the test scenario

This section describes the general characteristics of the scenario to continuously test the secure interface configuration of cloud service components.

5.5.2.1 Supported cloud-specific controls

The control *IVS-06: Infrastructure & Virtualization Security Network Security* of CSA's CCM [22] requires that

> *"Network environments and virtual instances shall be designed and configured to restrict and monitor traffic between trusted and untrusted connections. These configurations shall be reviewed at least annually, and supported by a*

[87] `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html` [Accessed: 2018-12-13]

[88] `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.RDSSecurityGroups.html` [Accessed: 2018-12-13]

documented justification for use for all allowed services, protocols, ports, and by compensating controls.".

The test CT^{SI} can support certification of cloud services according to this control because it helps to detect whether virtual instances are configured to restrict traffic between untrusted connections. Note that this control itself demands to be checked at least every year. The CCM links to the control *A.9.1.2 Access to networks and network services* of ISO 27001:2013 [24] which can also be checked using CT^{SI}. Furthermore, the control *IVS-07: Infrastructure & Virtualization Security OS Hardening and Base Controls* of CCM [22] requires that

"Each operating system shall be hardened to provide only necessary ports, protocols, and services to meet business needs [...].".

This control is partly very similar to the previous one since it also requires that only necessary ports, protocols, and services are exposed by the operating system of a dedicated server or virtual machine. It is evident that, through continuously testing for blacklisted ports, CT^{SI} can provide evidence to support checking if this control is satisfied.

In [253], the BSI links the control *IVS-07* of the CCM to the control *RB-22 Handling of vulnerabilities, malfunctions and errors – system hardening* of the BSI C5 [31] which demands that

"System components which are used for the rendering of the cloud service are hardened according to generally established and accepted industry standards. The hardening instructions used are documented as well as the implementation status.".

This control is less specific than the previous two examples from the CCM, thus it includes those two controls. Therefore, we can state that the certification of a cloud service according to *RB-22* of BSI C5 can be supported by CT^{SI}.

5.5.2.2 Supported cloud service models

As already mentioned in Section 5.5.1, CT^{SI} can be used to continuously check whether interfaces of PaaS applications such as Amazon RDS are correctly configured, that is, are only accessible from authorized IP addresses. Naturally, continuously probing for open ports is also suited to check if the security groups, i.e., firewall rules which are set for virtual machine instances (IaaS) are correctly configured.

5.5.2.3 Implementation independence

The continuous test CT^{SI} repeatedly probes a host or IP address for open ports. Therefore, it operates independent of the specific composition of the cloud service under test such as the operating system and other applications used to provide a particular cloud service.

5.5.2.4 Minimally invasive integration

CT^{SI} does not require any change to the composition of the cloud service under test. As with previously presented continuous tests, however, one prerequisite of CT^{SI} is that it has permission to reach the target host. Otherwise, it cannot conduct an analysis of the exposed interfaces of the cloud service's endpoint.

5.5.3 Test design

In this section, we describe the test CT^{SI} according to the framework introduced in the Chapter 4. Note that the configuration of CT^{SI}, that is, concrete values such as test suite iterations serve as examples and can vary with differing use cases of the test as well as the interpretation of the controls whose validation it seeks to support.

5.5.3.1 Test case

The test CT^{SI} consists of the following single test case:

$$\begin{aligned} TC^{Port} =& \langle E, L, O, N \rangle \\ =& \langle \textit{scan_ports} \rangle, \\ =& \langle \langle \textit{host, port_range, whitelisted_ports} \rangle \rangle, \\ =& \langle \textit{assert_whitelisted_ports_only} \rangle, 1 \rangle. \end{aligned}$$

Having a single procedure (E), the test case TC^{Port} probes a target host of the cloud service for open ports. To do this, the test case requires as input parameters (L) the IP address or hostname (*host*), the port range to scan (*port_range*), and list of permitted ports (*whitelisted_ports*). In order to evaluate the results produced by the test case (O), we check whether any other open ports than the whitelisted ones are detected. Thus, an instance of TC^{Port} passes if only ports are detected which are on the whitelist. Finally, the ordering number of the test case is 1.

5.5.3.2 Preconditions

In order to avoid false negatives, that is, test suite run results which incorrectly indicate that other than whitelisted ports are publicly available, we define the following precondition:

$$\begin{aligned} TC^{\widehat{ICMP}} =& \langle E, L, O, N \rangle \\ =& \langle \langle \textit{measure_delta_ICMP_Echo_Request_to_ICMP_Time_Reply_packets} \rangle, \\ & \langle \langle \textit{host, packet_count} \rangle \rangle, \\ & \langle \textit{assert_rtt_avg} < \$^{ICMP}_{avg}, \textit{assert_rtt_sd} < \$^{ICMP}_{sd} \rangle, 1 \rangle. \end{aligned}$$

Similar to the testing the secure communication configuration described in the previous section, the test CT^{SI} defines a precondition which tests if it can reach the target host on the Internet Layer. Also for this test, we re-use the test case definition TC^{ICMP} specified to continuously test resource availability as described in Section 5.2. However, in this test scenario we use $\widehat{TC}^{ICMP}$ to ensure that we can reach the host whose open ports we aim to identify with the main test suite. Since this precondition possesses ordering number 1, it will be executed first once the test suite it is bound to executes.

5.5.3.3 Test suite

The test CT^{SI} consists of the following, single test suite:

$$\begin{aligned} TS^{\langle Port, \widehat{ICMP} \rangle} =& \langle \mathcal{TC}, I, F, T \rangle \\ =& \langle \langle TC^{Port}, TC^{\widehat{ICMP}} \rangle, +\infty, 0, \langle 30 \rangle \rangle. \end{aligned}$$

The test case TC^{Port} and the precondition $TC^{\widehat{ICMP}}$ are bound to the test suite $TS^{\langle Port,\widehat{ICMP}\rangle}$. This means we are using the option *preconditions as part of main test suites* of our framework (for further details see Section 4.3.6) to establish that the test environment meets our expectations. The number of successive executions (I) is set to $+\infty$ which means that successive execution of the test suite can be triggered infinitely often. The interval (T) between each test suite run is set to 30 seconds, i.e., the next test suite execution is triggered 30 seconds after the previous one completed. Lastly, *no* offset (F) is defined for this test suite.

5.5.3.4 Workflow

The workflow of CT^{SI} is shown in Figure 5.11: It triggers execution of $TS^{\langle Port,\widehat{ICMP}\rangle}$ after the specified interval between test suites has elapsed. As indicated by the heavy bar, execution of the test suite translates to triggering execution of the test cases TC^{Port} and $TC^{\widehat{ICMP}}$ concurrently and joining them afterwards, i.e., the test suite completes if both test cases complete. The workflow is basic, i.e., the result of the test suite does not affect the repeated execution of test suite $TS^{\langle Port,\widehat{ICMP}\rangle}$.

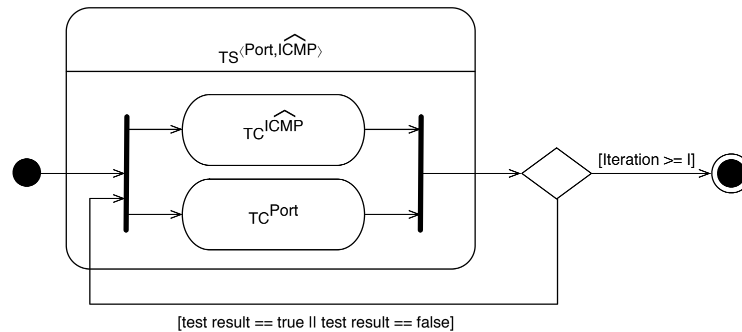


Figure 5.11: Workflow of continuously testing secure interface configuration (CT^{SI})

5.5.3.5 Test metric

The goal of the test CT^{SI} is to measure how long it takes to fix a temporarily incorrectly configured interface, that is, how long does it take to close supposedly blacklisted ports which were identified as publicly reachable. An ideal choice for this task is the universal test metric $fpsD$ which was introduced in Section 4.3.5. Through determining the duration of a fps, the test metric $fpsD$ tells us for how long the configuration of a cloud service component was insecure.

Since we are using a precondition as part of the main test suites, we have to consider the result of the precondition test results during computation of test metrics. Within this test scenario, execution of the precondition test case $\widehat{TC}^{ICMP}$ is triggered concurrently with the main test case TC^{Port}. Thus, the result produced by a test suite run will only be considered if the precondition test case passed.

5.5.4 Implementation and experiment

This section describes implementation as well as experimental evaluation of our approach to continuously test the secure interface configuration of cloud service components.

5.5.4.1 Environment and setup

Hereafter, the main components of the experiment are outlined.

Cloud service component under test As an example of a cloud service component under test, we choose an instance of MongoDB[89] which is one example of a NoSQL database. This database is deployed on an instance provided by OpenStack Mitaka with 2 vCPUs, 4 GB RAM and 40 GB volume (disk) running Ubuntu 15.10 Server. The instance is bound to a publicly reachable IP address while its security group is set to allow for traffic via ICMP, via TCP on port 22 and via SSH from and to the host where the test is deployed (see paragraph *Deployment of continuous test* below). This cloud service component is considered to be part of a cloud service of type PaaS.

Test case implementation In order to implement the test case TC^{Port}, we use Nmap[90], a popular tool which allows discovering hosts in a network as well as services provided by these hosts. The basic use of Nmap only requires to specify the hostname or IP address of one or more scan targets, e.g., *scanme.nmap.org* or 10.244.250.123 or both. Once the scan is completed, Nmap returns the results, e.g., as a XML-formatted document. A common use case of analyzing these results consists of manual inspection, for example, by a human security analyst. Yet, the test case TC^{Port} parses the results returned by Nmap to determine whether only whitelisted ports of the scan target can be reached. If other than the whitelisted ports can be reached, then the TC^{Port} fails, otherwise it passes.

Precondition implementation In order to implement the precondition $\widehat{TC}^{ICMP}$, we reuse the implementation of the test case TC^{ICMP} of the test CT^{AV} described in Section 5.2.4.1. The right-hand side of the assertions of the oracle of $\widehat{TC}^{ICMP}$ are specified as follows:

- $\$^{ICMP}_{avg}$ = 100ms, and
- $\$^{ICMP}_{sd}$ = 75ms.

Deployment of continuous test The test CT^{SI} is deployed on a different host, attached to a different network than the host of the cloud service component under test. Yet both hosts are located in the same building. As a result, delays on the IP Layer observed prior to the experiment are relatively low, i.e., usually lower than 3ms.

Inducing insecure interface configurations In order to trigger temporary insecure interface configuration of our cloud service component under test, we edit the configuration file of the MongoDB server making port 27018 publicly reachable. 27018 is the default port for sharded[91] instances of MongoDB that should not be publicly reachable. In total, we triggered 1000 vulnerable interface configuration events where each event lasted at least 180 seconds plus selecting [0,60] seconds at random. Further, the interval between consecutive

[89] `https://www.mongodb.com/` [Accessed: 2018-12-13]

[90] `https://nmap.org/` [Accessed: 2018-12-13]

[91] *Sharding* is a technique which horizontal splits a data base into partitions each called a *shard* and kept on an individual instance of the database server [278]. MongoDB uses *sharding* to handle very large data sets as well as high throughput operations. For further information see `https://docs.mongodb.com/manual/sharding/`.

vulnerable interface configuration events was at least 200 seconds plus selecting [0,100] seconds at random.

The accumulated duration of all vulnerable interface configuration events observed during the experiment is 209747.99 seconds (≈ 58 hours 15 minutes 48 seconds). Further, the mean duration of each downtime event is 209.75 seconds with a standard deviation of 17.67 seconds.

5.5.4.2 Experimental results

The entire experiment lasting from the start of the first test until the end of the last test took ≈127,9 hours. Table 5.4 shows the test statistics and test results produced by the test CT^{SI}. The total number of tests, i.e., the total number of executions of the test suite $TS^{\langle Port,\widehat{ICMP}\rangle}$ is 32589. On average, each execution of a test suite of CT^{SI} took ≈9 seconds, with a standard deviation of 0.04 seconds.

Table 5.4: Test statistics and results of continuous test CT^{SI}

Test statistics	number of tests	32589
	mean duration test (sec)	9.06
	sd duration test (sec)	0.04
	min duration test (sec)	9.01
	max duration test (sec)	10.13
fpsC	correctly detected events	986
	correctly detected events (%)	98.6%
fpsD	mean duration detected events (sec)	209.28
	sd duration detected events (sec)	18.52
	min duration of detected events (sec)	141.89
	max duration of detected events (sec)	252.99

CT^{SI} correctly detected 986 vulnerable interface events out of 1000 events, that is, 98.6% ($fpsC$). Moreover, CT^{SI} estimates that a vulnerable interface configuration lasts on average 209.28 seconds ($fpsD$) which means that CT^{SI} underestimates a vulnerability event on average by 0.47 seconds.

5.6 Continuously testing user input validation

In this section, we present the test CT^{UI} which aims at continuously checking whether a cloud service always validates user input. After having described the property *user input validation* in context of a cloud service and outlining the test scenario, we discuss the general characteristics of the test scenario (Section 5.6.2). Then we detail the design of CT^{UI} (Section 5.6.3) and, finally, describe its implementation as well as experimental results (Section 5.6.4). Parts of this section have been published in [249].

5.6.1 Property description and overview of test scenario

As already described as part of the background on cloud services (see Section 2.1.1), 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 [10]. 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 [279].

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 [280]. 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 type 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 [281].

Consider, as an example, that at some point in time auditing a SaaS application reveals that it possesses SQL injection vulnerabilities. Let's assume that, as a reaction, data sanitization is implemented at the database layer using stored procedures which depicts one possible countermeasure. However, if this SaaS application makes use of frameworks such as Ruby on Rails[92], then changing the database used by the application's controller is achieved through simple configuration changes. In case the newly deployed database 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[93]. As a result, another layer of abstraction is added to the architecture of the SaaS application where changes in the back end rendering the SaaS application vulnerable are hard to detect, even for the SaaS provider herself.

Checking SQLI vulnerabilities of SaaS applications thus requires an approach capable of continuously, i.e., automatically and repeatedly check whether the cloud service validates user input. To that end, we propose the test CT^{UI} which continuously tests web application components of a SaaS application for SQLI vulnerabilities.

In summary, this test work as follows (a detailed description of the test will be provided in Section 5.6.3): Every 30 seconds, it uses a URL of the cloud service under test to scan 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 CT^{UI} aim at counting the times during which the cloud service is vulnerable to SQLI within a particular period of time.

5.6.2 General characteristics of the test scenario

This section describes the general characteristics of the scenario to continuously test user input validation of SaaS application components.

[92] `http://rubyonrails.org/` [Accessed: 2018-12-13]
[93] `https://cloud.google.com/appengine/` [Accessed: 2018-12-13]

5.6.2.1 Cloud-specific controls

The control *RB-21: Handling of vulnerabilities, malfunctions and errors – check of open vulnerabilities* of BSI C5 [31] requires that

> *"The IT systems which the cloud provider uses for the development and rendering of the cloud service are checked automatically for known vulnerabilities at least once a month.*
> *In the event of deviations from the expected configurations (for example, the expected patch level), the reasons for this are analyzed in a timely manner and the deviations remedied or documented according to the exception process (see SA-03).".*

The test CT^{UI} can support the certification of a cloud service according to this control because it automatically checks if a cloud service has known SQLI vulnerabilities. Note that *RB-21* itself demands that testing for vulnerabilities should be at least conducted monthly.

Another control whose evaluation can be supported by CT^{UI} is *TVM-02: Threat and Vulnerability Management Vulnerability / Patch Management* of CSA's CCM [22]. The BSI maps its own control *RB-21* to *TVM-02*. One part of the latter one requires that

> *"Policies and procedures shall be established, and supporting processes and technical measures implemented, for timely detection of vulnerabilities within organizationally-owned or managed applications, infrastructure network and system components (e.g., network vulnerability assessment, penetration testing) to ensure the efficiency of implemented security controls. A risk-based model for prioritizing remediation of identified vulnerabilities shall be used [...].".*

Finally, both *TVM-02* of CCM and *RB-21* of BSI C5 map themselves to the control *A.12.6.1 Management of technical vulnerabilities* of ISO/IEC 27001:2013 [24]. The last two controls, i.e., *TVM-02* of CCM and *A.12.6.1* of ISO/IEC 27001:2013 go beyond mere testing for vulnerabilities of an information system or cloud service by also requiring risk-driven remediation of detected vulnerabilities. However, both include *timely detection* of vulnerabilities a cloud service may have which can be supported by the test CT^{UI}.

5.6.2.2 Supported cloud service models

As already pointed out in the introduction of this test scenario, the test CT^{UI} can be used to continuously check if SaaS applications are vulnerable to SQLI, that is, if a SaaS application validates user input. It cannot be used with PaaS applications or IaaS because neither of these provide their services through a web browser.

We note that providers of IaaS and PaaS applications may offer to manage their services through browsers, e.g., Amazon's AWS Management Console. Such administration interfaces are web application which can be considered SaaS applications. However, the actual cloud service provided by IaaS and PaaS providers is not used by customers through a browser.

5.6.2.3 Implementation independence

The test CT^{UI} can be used with any SaaS application, that is, the part which uses web application technology and has some means to accept user input, e.g., text fields.

Further, CT^{UI} is agnostic to specific database back ends such as PostgreSQL[94], MySQL[95], MariaSQL[96] or SQLite[97] which process user input provided to a SaaS application. All of these SQL database implementations may possess some known SQLI vulnerability which CT^{UI} aims to detect.

5.6.2.4 Minimally invasive integration

Although it is a possibility to deploy the implementation CT^{UI} as an additional component of the SaaS application, this is not required in order for CT^{UI} to produce test results. If the SaaS application's web server can be reached from any remote host, then no change of the SaaS application's configuration or composition is required in order for a remotely deployed CT^{UI} to work correctly. However, if the access to the application is somehow restricted, for example, if it is only accessible within an internal network, then the host on which the implementation CT^{UI} is deployed has to be granted access to that network as well. Similarly, if firewall rules limit access to an IP address range, then traffic from the IP from which CT^{UI} sends requests to the SaaS application has to be permitted by the firewall.

5.6.3 Test design

In this section, we describe the test CT^{UI} according to the framework introduced in the Chapter 4. Note that configurations of CT^{UI} described hereafter are example values (e.g., the value configured for test suite offset) which can vary with the concrete test's use case and the interpretation of the controls whose validation it aims to support.

5.6.3.1 Test case

The test CT^{UI} consists of only one test case which is defined as follows:

$$\begin{aligned} TC^{SQL} &= \langle E, L, O, N \rangle \\ &= \langle \textit{scan_target_url} \rangle, \\ &= \langle \langle \textit{target_url, session_cookie, target_data} \rangle \rangle, \\ &= \langle \textit{assert_empty_scan_results} \rangle, 1 \rangle. \end{aligned}$$

The test case TC^{SQL} uses only a single procedure (E) which scans a target URL for SQLI vulnerabilities. To that end, the test case needs the input parameters (L) *target_url*, *session_cookie*, and *target_data*. The cookie serves to provide some valid session information if required, thus allowing the scan for SQLI vulnerabilities to be executed within a particular user session which potentially has access to other fields than a user who is not logged in. Further, the parameter *target_data* specifies which information a scan should retrieve in case a SQLI is found, e.g., list of database users or list of tables. In order to evaluate the results produced by the test case, we define a single oracle (O) using the assertion *assert_empty_scan_results*. Therefore, an instance of TC^{SQL} passes if a scan does not return any results, that is, does not retrieve any specified target data. Lastly, the ordering number (N) of TC^{SQL} is 1.

[94] `https://www.postgresql.org/` [Accessed: 2018-12-13]
[95] `https://www.mysql.com/` [Accessed: 2018-12-13]
[96] `https://mariadb.org/` [Accessed: 2018-12-13]
[97] `https://www.sqlite.org/` [Accessed: 2018-12-13]

5.6.3.2 Test suite

The user input validation test consists of the following single test suite:

$$
\begin{aligned}
TS^{\langle SQL \rangle} =& \langle \mathcal{TC}, I, F, T \rangle \\
=& \langle \langle TC^{SQL} \rangle, +\infty, 2, \langle 30 \rangle \rangle.
\end{aligned}
$$

The single test case TC^{SQL} of CT^{UI} is bound to the test suite $TS^{\langle SQL \rangle}$. Its number of successive execution (I) is set to $+\infty$ indicating that test suite's execution can be triggered infinitely unless interrupted by the workflow. Further, execution of this test suite is triggered statically every 30 seconds (T) after the previous test suite run completed. Lastly, there is an additional two seconds offset (F) between successive test suite runs.

5.6.3.3 Workflow

Since CT^{UI} only consists of one test suite, the workflow we use to control the execution of test suite runs is the most basic one (Figure 5.12). It simply triggers execution of $TS^{\langle SQL \rangle}$ after the specified interval between test suite runs has elapsed. This means that the result of the test suite does not affect the repeated execution of test suite $TS^{\langle SQL \rangle}$.

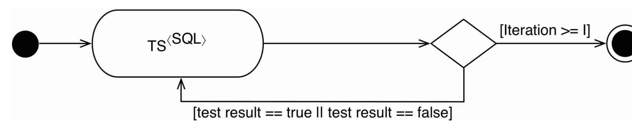


Figure 5.12: Workflow of continuously testing user input validation (CT^{UI})

5.6.3.4 Test metric

The continuous test CT^{UI} aims at determining the number of times a cloud service of type SaaS was vulnerable to SQLI within a particular period of time. To that end, we draw on the universal metric $fpsC$ which counts the occurrences of Failed-Passed-Sequences (fps). In context of our test CT^{UI}, a single fps is the time where execution of TC^{SQL} failed at least once and then passed, that is, the scanning of the target URL revealed SQLI vulnerabilities at some point in time which were fixed later on. Now these vulnerabilities may persist for some time which, in turn, leads to multiple, successive failed test suite runs. Once the test suite run passes, i.e., the SQLI vulnerabilities were fixed, a single instance of test metric fps is produced. Counting any observed fps thus gives us the times during which the SaaS application was vulnerable to SQLI.

5.6.4 Implementation and experiment

This section describes implementation as well as experimental evaluation of our approach to continuously test user input validation of SaaS application components.

5.6.4.1 Environment and setup

Hereafter, the main components of the experiment are outlined.

Cloud service component under test We select an instance of the Damn Vulnerable Web Application (DVWA)[98] which is deployed on an instance of OpenStack Mitaka with 2 vCPUs and 4 GB RAM and 40 GB volume (disk) running Ubuntu 15.10 Server as an example of a SaaS application component. Moreover, the instance is associated with a publicly reachable IP address and its security groups are configured to permit traffic via HTTP and via SSH from and to the host where the continuous test is deployed (see paragraph *Deployment of continuous test* below).

Furthermore, DVWA is a PHP-based web application which is intentionally set up to be vulnerable to different types of attacks, including SQLI. It is usually used for educational purposes, giving students the opportunity to learn how vulnerabilities of web applications can be exploited. To that end, vulnerabilities of the DVWA are configurable, i.e., can be set according to the required sophistication needed to be successfully exploited.

Test case implementation We use SQLMap[99] to implement the test case TC^{SQL}, a popular tool specifically and solely targeting SQLI discovery. It supports various attack methods[100] including UNION query-based, stacked queries, out-of-bound, boolean-based blind, time-based blind and error-based attacks.

SQLMap works as follows: It is provided a URL which serves as the attack target, in our case a URL where the SaaS application can be reached. SQLMap then tries to guess which back end database management system (DBMS) the SaaS application uses. Based on this guess, SQLMap attempts to attack the DBMS using attack vectors specific to the presumed DBMS. In case of a successful attack, SQLMap can fetch data from the compromised database, for example, users, the name of current database in action, or the schema of database.

SQLMap is usually used interactively by a human user, e.g., a penetration tester, who operates the tool and evaluates its output. However, it also exposes most of its features through a RESTful API which we use to implement the test case TC^{SQL}. More specifically, we developed a library which is used by TC^{SQL} to call the SQLMap RESTful endpoints and parse as well as evaluate the output returned by SQLMap. The following API calls are required to execute a vulnerability scan: First, a new SQLMap scan task is initiated, then the options for that task are set, and thereafter the scan execution is triggered. While the scan is running, we periodically check whether the scan is finished and, if so, fetch the result produced by SQLMap. Finally, the scan task is deleted to free system resources bound to the scan task by SQLMap. Lastly, scanning multiple endpoints of a SaaS application is also supported by TC^{SQL} through assigning each scan a unique ID.

In case of successful exploitation of the SaaS application, then SQLMap returns the current DBMS user, a list of tables and a list of database users. This means that the SaaS application is vulnerable to SQLI and thus the test fails. Otherwise, if SQLMap does not return any of the above information, then we consider the SaaS application not to be vulnerable and the test passes.

Deployment of continuous test The test CT^{UI} is deployed on a different host which is attached to a different network than the host of the service component under test. However, both hosts are located in the same building.

[98] `http://www.dvwa.co.uk/` [Accessed: 2018-12-13]

[99] `https://www.sqlmap.org` [Accessed: 2018-12-13]

[100] For a comprehensive introduction into different SQLI attack types see, e.g., Halfond et al. [281].

Inducing SQLI vulnerability events In order to trigger temporary vulnerabilities of the DVWA which our test CT^{UI} aims to detect, we edit the configuration file of DVWA, thereby rendering it vulnerable for some predefined period of time. Then, we change the configuration file of the DVWA again, this time making it secure against SQLI. In total, we trigger 500 sequential events during which the DVWA is vulnerable to SQLI. Each of these vulnerability events lasts at least 60 seconds and an additional [0, 10] seconds selected randomly per event. Further, the interval between consecutive vulnerability events is at least 60 seconds plus selected another [0, 30] seconds at random. The accumulated duration of all vulnerability events was 32474.68 seconds (≈ 9 hours 1 minute 14 seconds), mean duration of each vulnerability events was 64.95 seconds with a standard deviation of 3.11 seconds.

5.6.4.2 Experimental results

Table 5.5 shows the results which CT^{UI} produced: The total number of times the test suite $TS^{\langle SQL \rangle}$ was executed is 1337. Each execution of a test suite or a test took 20.25 seconds on average with a standard deviation of 19.54 seconds. Out of the 500 temporary vulnerability events of the DVWA, the test CT^{UI} correctly detected 395 or 79%.

Table 5.5: Test statistics and results of continuous test CT^{UI}

number of tests	1337
mean duration test (sec)	20.25
sd duration test (sec)	19.54
min duration test (sec)	6.02
max duration test (sec)	101.14
number of induced events	500
number of detected events	395
detected events (%)	79

5.7 Summary and discussion

In this chapter, we presented five example test scenarios to show how the framework introduced in Section 4.3 supports continuous certification of cloud services. These scenarios exemplify how to design and implement continuous tests of the following cloud service properties:

- Availability,
- location,
- secure communication configuration,
- secure interface configuration, as well as
- user input validation.

We explained how testing these cloud service properties can support continuously certification according to example controls selected from

- the Cloud Computing Compliance Controls Catalogue (BSI C5),

- the Cloud Control Matrix (CCM) of the Cloud Security Alliance (CSA), and
- ISO/IEC 27001:2013 published by the International Organization for Standardization (ISO).

At this point, a legitimate criticism of such a scenario-driven approach is if it may fall short with regard to completeness and thus may be insufficient to demonstrate the applicability of our framework in general. In order to counter this argument, we identified four general characteristics to describe continuous test scenarios. These characteristics allowed us to reason about the applicability of our framework beyond the scope of the example scenarios. These characteristics include:

- Plausible link to cloud-specific controls,
- supported cloud service models,
- implementation independence, and
- minimally invasive integration.

We described each of the five example test scenarios according to these characteristics. That way, we are able to compensate for potential shortcomings of using a scenario-driven approach to demonstrate the applicability of our framework.

We note that our approach does not guide the selection of those cloud service properties which are *most relevant* in context of a particular scenario. Having such a method would allow designing and implementing test scenarios which are bound to real-world use cases and, therefore, further support demonstration of our framework's applicability. In this context, the possibility to design scenario-specific metrics using our framework could be shown, e.g., by implementing a metric which quantifies the *security level* of a TLS configuration within the *secure communication configuration* scenario.

Furthermore, we note that only considering minimally invasive test designs has limitations as to which data, that is, which evidence can be produced and analyzed. As a result, the quality of statements and possibly also the scope of controls which can be evaluated using minimally invasive tests is limited as well. However, we do not provide a method to formalize these restrictions which would help reasoning about the applicability of minimally invasive continuous tests in general, independent of concrete test scenarios. Developing such an approach is not in scope of this thesis and thus subject to future work.

Chapter 6

Definition of continuous tests

In the previous chapters, we described a framework to design tests which supports continuous test-based certification of cloud services (Chapter 4) and demonstrated its application (Chapter 5). This framework does not mandate a particular architecture prescribing how to implement continuous tests. Yet when using tests to automatically and repeatedly produce evidence, it is vital to *unambiguously define how evidence is produced* to check one or more controls at a certain point in time; only then can we rigorously compare the evidence produced by different test implementations. In order to guide development of tests according to our framework, this chapter introduces an approach to rigorously define such tests. This addresses *Research challenge 2: Definition of continuous tests* described in Section 1.2.2.

A method to ensure rigorous definition of continuous tests can make use of *formal languages* because these languages are based on precise mathematical definitions [282]. Drawing on that notion, this chapter presents a domain-specific language (DSL) called *ConTest*. ConTest is a descriptive language which, on the one hand, is agnostic to a specific implementation of the framework's building blocks introduced in the Section 4.3. One example implementation of the framework is *Clouditor's engine* which is described in Section 4.4. On the other hand, ConTest serves as a starting point to generate concrete configurations for any specific implementation of a continuous test, for example, test configurations for Clouditor's engine. That way, ConTest serves as a unified configuration language ensuring that the configuration of a continuous test implementation adheres to the building blocks defined in Section 4.3.

The next section outlines concepts needed to develop domain-specific languages. Thereafter, in Section 6.2, the DSL engineering process described by Mernik et al. [283] is followed to develop ConTest (an outline of this process is described in Section 6.1): First, the decision why to develop a DSL is explained (Section 6.2.1) followed by the analysis of the required constructs (Section 6.2.2) which draws on the build blocks defined in Section 4.3. On this basis, Section 6.2.3 formally defines the grammar of ConTest using Extended Backus-Naur Form (EBNF). Using this formal definition, Section 6.2.4 describes the implementation of ConTest using the language development tool XText[101] and shows how to use code generators to translate from a test definition written in ConTest to a Clouditor test configuration, one example of a concrete test configuration language. Finally, Section 6.3 summarizes the contents of this chapter and discusses the approach.

[101]`https://eclipse.org/Xtext/` [Accessed: 2018-12-13]

6.1 Developing domain-specific languages

As the name indicates, domain-specific languages (DSLs) are specialized languages which are tailored to a particular application domain [283]. This specialization comes at a price: A DSL trades flexibility of general purpose languages (GPLs) to express *any* program for productivity and conciseness to construct programs which solve a domain-specific set of problems. To that end, DSLs provide features that are optimized to solve such domain-specific problems [284].

Yet a clear distinction between DSLs and GPLs is not trivial. Some attempts to measure the specificity of a language have been made [285], providing intuitive results such as Backus-Naur Form (BNF) and C++ each are rather at the end of the spectrum where Cobol would be closer to C++ than to BNF. Other approaches argue that domain-specificity of languages is rather gradual and rely on a qualitative distinction based on criteria such as *language size* or *Turing Completeness* [284].

The motivation why to use a DSL varies. Usually, the aim is to improve productivity and reduced maintenance compared to using GPLs. This has been shown to be true in some cases, e.g., by [286] and [287]. Another driver is that a DSL can reduce the amount of domain and programming expertise required, thereby making the application domain accessible to a larger group of developers [283].

The question at this point is what advantages does a DSL provide to define continuous tests? The answer is that it permits us to choose a level of abstraction which allows us to specify all important parts of a test while – at the same time – it is agnostic to a particular implementation of the building blocks of the framework described in Section 4.3. Thus, a DSL provides strict rules to define a continuous test in general while a developer seeking to implement a continuous test can use the DSL as a starting point to identify constructs necessary to configure the specific continuous test.

But how do we define a suitable DSL? We will address this question in the following two sections: First, we will describe the DSL engineering process as proposed by Mernik et al. [283] (Section 6.1.1). Thereafter, we will outline the core concepts of formal languages, explain context-free grammars and introduce EBNF as a syntax specification formalism (Section 6.1.2).

6.1.1 DSL engineering

There are multiple steps involved when developing a domain-specific language. In the following, we describe the steps involved in DSL engineering which were proposed by Mernik et al. [283]:

1. *Decision:* First, developing a new DSL or identifying an existing DSL to reuse has to be properly motivated because it initially incurs additional effort. Such motivating factors can be driven by 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.

2. *Analysis:* In this step, the domain for the which the DSL should be developed is identified, scoped, and described. To that end, inspection of existing GPL code, technical documentation, and interview with domain experts can provide the needed input for analysis. The result of this analysis is a description of the domain-specific terminology and semantics.

3. *Design:* When designing the DSL, there are two major approaches which can be followed: The first one is to base the DSL on an existing language where we can either use some features of the existing language (*piggyback*), restrict the existing language (*specialization*) or extend the existing language (*extension*). The second option to design a DSL is to build it from scratch, i.e., the design does not relate to any existing language.

   Having decided whether to invent a new language or to build on an existing one, the next step consists of defining the language. This can be done formally or informally. In the latter case, natural language can be used to describe the features of the DSL to be developed. However, this approach will not suffice if the goal is to build a DSL that can actually be consumed by an application. Thus we need to formally describe the syntax of the DSL which can be achieved using, e.g., EBNF, which we will describe in Section 6.1.2.

4. *Implementation:* Having designed a DSL, the question in this step is how to implement the DSL. According to Mernik et al. [283], this include, for example, selecting one of the following approaches:

   - *Compilers* which translate the DSL constructs to an existing language's constructs 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 whereas an *external* DSL is represented in a language different to main programming language it is interacting with [288].

   Finally, implementing a DSL using a compiler or interpreter has many advantages, e.g., the syntax can be close to notations used by domain experts. Yet it also bears disadvantages such as a complex language processor may have to be implemented. However, these disadvantages can be limited or eliminated if language development tools are used which automate most of the language processor construction [283]. Examples for such tools are XText[102], Spoofax[103] or MPS[104].

Now that we outlined the main steps of the DSL engineering process, the next section introduces the necessary concepts of formal languages which are required to formally define a DSL.

6.1.2 Formal languages

A formal language L is defined by an alphabet Σ and a grammar G. The alphabet Σ is a set whose elements are called *symbols*. A finite sequence of symbols from Σ are called *word*.

[102] `https://eclipse.org/Xtext/` [Accessed: 2018-12-13]
[103] `https://www.metaborg.org/` [Accessed: 2018-12-13]
[104] `https://www.jetbrains.com/mps/` [Accessed: 2018-12-13]

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 $\langle N, T, R, S\rangle$ where

- N is the set of nonterminals, i.e., variables that represent language constructs,
- T is the set of terminals which is identical with the alphabet Σ 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 $l \to r$. Both l and r are sequences of nonterminals and terminals with l containing at least one nonterminal, and
- S is a nonterminal, i.e., $S \in N$ which constitutes the start variable.

6.1.2.1 Extended Backus-Naur Form

The Backus-Naur Form (BNF) is a technique to define context-free grammars [289]. The BNF can be viewed as a domain-specific language itself which was developed to ease syntax specification [283]. Building on the BNF, the extended Backus-Naur Form (EBNF) [290] has been developed and standardized in ISO/IEC 14977:1996 [291]. As an example, consider the following context-free grammar G of the language $L = \{a^k b^k | k \geq 1\}$:

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

Thus deriving the word $\hat{w} = aaabbb$ works as follows:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb.$$

When using BNF to represent the grammar of this language, we first have to lay out some syntactic conventions: Usually, the symbol '::=' is used for productions instead of '→'. Furthermore, the symbol '|' is used to represent alternative derivations in a more efficient way 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

$$\langle S\rangle ::= 'a'S'b'|'a''b'.$$

The EBNF further improves efficiency of defining context-free grammars. It should be noted that – despite the standardization effort in ISO/IEC 14977:1996 – there is no universally accepted variant of EBNF. We choose a version whose syntax is very similar to the one used by *xText*[105], an open source framework to implement domain-specific languages, which we will use later to implement our test definition language (see Section 6.2.4). Also, this EBNF version is used to define the formal grammar of XML [292]. The most important extensions to syntactic conventions of BNF are the following:

[105] `https://eclipse.org/Xtext/` [Accessed: 2018-12-13]

- Nonterminal symbols are *not* enclosed with angle brackets (as 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:

$$S ::= aS * b.$$

6.2 Continuous test definition language

This section describes the development of a domain-specific language called *ConTest*. This language aims at supporting the rigorous, unified definition of continuous tests. The structure of this section follows the DSL engineering process described in Section 6.1.1.

6.2.1 Decision

As already pointed out in the beginning of Chapter 6, the motivation to develop a DSL to define continuous tests lies in having an approach at hand to specify all required parts of such a test in general. This not only paves the way for comparability of evidence produced by different test implementations but also guides the development of test implementations: If a developer decides to design and implement a continuous test, then she uses ConTest to make sure that the configuration of the test implementation indeed adheres to the building blocks introduced in Section 4.3.

Since the DSL is defined by a formal grammar, the guarantee of a test implementation conforming with the framework is not merely informal but is enforced through code generators: The developer has to provide a code generator which translates the constructs of ConTest into the target language constructs, that is, the language the specific implementation uses to configure a continuous test. Put differently: Any specific test implementation has to be configurable using a test definition written in ConTest. This implies that a suitable application generator exists which translates the constructs of ConTest into the language constructs which a particular test implementation uses.

6.2.2 Analysis

In order to identify the required constructs which ConTest should provide, we draw on the description of the building blocks of our framework to support continuous test-based cloud service certification described in Section 4.3:

- *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 a descriptive test definition language is unnecessary

because the implementation of the procedures is left the developer implementing the test. Rather, we summarize all procedures of a test case by a construct named *TestCaseModule* which points to a particular component of the test which implements the necessary procedures.

Regarding the *list of input parameters* which are used as input to the procedures, we include those in the test definition language. A test implementation 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, we do not include a procedural description of the oracle in the test definition language since the actual evaluation of test case results is left to the 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, we include a list of assert parameters which a test implementation passes to the test oracle.

The ordering number is used to prioritize test cases' execution as part of a test suite. The ordering number is included in the test definition language.

Finally, each instance of a test case that is specified as part of a continuous test definition has to be addressable through a unique ID (unique in scope of the test definition instance).

- *Test suite:* One or more test cases ($\mathcal{TC}$) 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 test definition. We described which part of a test case has to be represented by the test definition above. In order to bind a test case to a test suite within a continuous test definition, the unique ID of a test case is used.

  The *iterations* of a test suite, that is, how many times the test suite can be successively 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 test definition. Also the *interval* between two text suite executions is specified as part of the test definition 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 depending on the current iteration.

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

- *Workflow:* Deciding what test suite to execute next is the responsibility of the workflow. A continuous test uses exactly one workflow.

  A test definition does not include the procedural elements of a workflow but only a pointer to the *WorkflowModule* which a specific test implementation uses. Furthermore, a test definition includes the test suites which a workflow may use. To that end, defined test suites are bound to the workflow using their unique ID.

- *Test metrics:* Test metrics allow us to reason about a sequence of test results produced by a continuous test. A continuous test may compute one or more test metrics. The actual

procedures which a test metric may use are not included in the continuous test definition because those are specific to a test implementation. Similar to a workflow definition and test case definitions, defining test metrics includes a pointer to *TestMetricModule*, i.e., the part of the test implementation where the test metrics are actually computed.

- *Precondition:* In order to test assumptions made about the environment of the cloud service under test, preconditions are used. Since preconditions can be either be designed as a specialized test suite or as precondition test cases (for further details see Section 4.3.6), no additional constructs for a continuous test definition are needed.

6.2.3 Design

The context-free grammar which generates ConTest is shown in Listing 6.1. 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 continuous test. Further, $'TestName'$ and $'TestDescription'$ each are followed by the variable $String$ providing a descriptive name and a brief description of the test, respectively. 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 implementation which implements the desired test metric, e.g., a particular Java class.

- *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 test implementation. Further, $Workflow$ is defined by one or more $TestSuite$ – following the terminal $'BoundTestSuites'$ – 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 either the variable Int which specifies the upper bound of successive iterations a particular test suite can be executed during a continuous test or by the terminal $'infinite'$ indicating that a testsuite's successive iterations are to set to (positive) infinity.

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 4.3.3: 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 depending on the current iteration of the test suite. Covering this case in ConTest, we use the terminal $'sequenceFixedInterval'$ with variable $ListInt$.

Further, $TestSuite$ is defined by the terminals $'Offset'$ and $'Timeout'$ each of which is followed by the variable Int. As described in Section 4.3.3, 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 by the workflow. This is particularly important if external tools such as *Nmap*[106] 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 the terminal $'BoundTestCases'$ which is followed by one or more $TestCase$. The definition of $TestCase$ is provided in the next 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 $'Order'$ followed by Int and the optional $'InputParameters'$ followed by the variable $Parameter$. This means that specifying input parameter for a test case are not required by any implementation of a test case. Further, the pointer to a particular test case implementation is assigned to the variable $String$ following 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 assert parameter is required since we need the assert parameter 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 $ListString$ or $ListInt$. $Parameter$ is defined by 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 4.3.2 where we introduced the concept of *oracles*, that is, methods determining whether a test case failed or passed. Thus $'AssertParameters'$ 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 to construct $String$, Int, ID, $KeyValue$, $ListString$, $ListInt$, $Range$ and $Parameter$ (Lines 48 - 62) which are primitive and composite data types of ConTest.

[106] `https://nmap.org/` [Accessed: 2018-12-13]

Listing 6.1: Context-free grammar of ConTest using Extended Backus-Naur Form (EBNF)

```
1  ConTest    ::=  Test
2
3  Test       ::=  'TestID' ID
4                  'TestName' String
5                  'Description' 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'
```

Listing 6.2 shows an example test definition of a continuous test which aims to test the availability of the load balancing component of a cloud service.

Listing 6.2: Example of a continuous test definition using ConTest

```
1
2  TestID d9ecdea3-e9da-4f78-b041-69b7509e3462
3  TestName "AvailabilityTestOfLoadBalancer"
4  Description "Continuously tests the availability of the load balancing
5              component of a cloud service"
6  Testmetrics{
7    TestMetricID YearlyAvailability{
8      TestMetricName "YearlyAvailabilityMetric"
9      TestMetricModule "de.fraunhofer.aisec.clouditor.metrics.YearlyAvailabilityMetric"
10     Description "Calculates the ratio between measured
11     downtime since test start and 31557600 seconds in a year
12     of 365.25 days"
13   }
14   TestMetricID DailyAvailability{
15     TestMetricName "DailyAvailabilityMetric"
16     TestMetricModule "de.fraunhofer.aisec.clouditor.metrics.DailyAvailabilityMetric"
17     Description "Calculates the daily availability starting
18                 from 00:00 am to 23:59 (UTC)"
19   }
20 }
21
22 Workflow {
23   WorkflowID PingTCPAvailabilityWorkflow
24   WorkflowName "BasicIterationWorkflow"
25   WorkflowModule "de.fraunhofer.aisec.clouditor.workflow.BasicIterationWorkflow"
26   BoundTestSuites {
27     TestSuiteID PingTCPAvailabilityTestSuite {
28       TestSuiteName "AvailabilityTestSuite"
29       NumberOfMaxIteration 1000000
30       IntervalBetweenTests {
31             randomizedInterval [30,60]
32       }
33       Offset 15
34       Timeout 600
35       BoundTestCases {
36         TestCaseID PingTest {
37           TestCaseName "PingTest"
38           TestCaseModule "de.fraunhofer.aisec.clouditor.testcases.PingTestCase"
39           Order 1
40           InputParameters {
41             <"count": 10>,
42             <"host": "10.244.250.9">
43           }
44           AssertParameters {
45             <"round-trip-avg-$lte":"50 ms">,
46             <"round-trip-sd-$lte":"25 ms">
47           }
48         }
49
50         TestCaseID TCPTest {
51           TestCaseName "TCPTestCase"
52           TestCaseModule "de.fraunhofer.aisec.clouditor.testcases.TCPTestCase"
53           Order 1
54           InputParameters {
55             <"probe":10>,
56             <"host" :"10.244.250.9">,
```

```
57              <"port" :22>
58           }
59           AssertParameters {
60              <"response-time-avg-$lte":"75 ms">,
61              <"response-time-max-$lte":"100 ms">
62            }
63          }
64        }
65      }
66    }
67 }
```

6.2.4 Implementation

In order to implement our DSL ConTest, we use the language development tool *XText*[107]. This tool is an open source framework which supports 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.

XText uses a proprietary language to specify the grammar of a DSL. However, the notation of this language is very similar to EBNF. Hereafter, we outline the characteristics of the language which XText uses to specify a grammar:[108]

- 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 6.1.2).
- 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)[109], XText stores parsed programs as in-memory object graphs. These graphs are instances of EMF Ecore models and represent the AST. Through using these structured data models, XText allows associating semantics of a meta-model which is accessible 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:

$$
\begin{aligned}
&TestSuite:\\
&\qquad 'TestSuiteID'\ tsrId = ID\\
&;
\end{aligned}
$$

[107] https://eclipse.org/Xtext/ [Accessed: 2018-12-13]
[108] https://eclipse.org/Xtext/documentation/301_grammarlanguage.html [Accessed: 2018-12-13]
[109] https://eclipse.org/modeling/emf/ [Accessed: 2018-12-13]

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 (which 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:

$$
\begin{aligned}
&TestCase: \\
&\qquad 'TestCaseID'\ name = ID \\
&\qquad 'TestCaseName'\ desc = STRING \\
&\qquad ; \\
&TestSuite: \\
&\qquad 'BoundTestCases'\ boundTestCases+ = [TestCase]+ \\
&\qquad ;
\end{aligned}
$$

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

Implementing ConTest with XText Listing 6.3 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 (Listing 6.1), then we can see that they are slightly different: In the case of the xText grammar, the variable $Test$ is also defined by at least one $TestSuite$ and by at least one $TestCase$ (Lines 15 to 19 of Listing 6.3) whereas in the EBNF representation, $TestSuite$ is part of the definition of the variable $Workflow$ (Line 22 of Listing 6.1) and $TestCase$ is part of the definition of $TestSuite$ (Line 36 of Listing 6.1).

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 continuous test, 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.

Listing 6.3: XText grammar definition for ConTest

```
grammar de.fraunhofer.aisec.conTestDSL.ConTestDSL with
org.eclipse.xtext.common.Terminals

ConTest:
        test=Test
;

Test:
    'TestID' name=ID
    'TestName' testName=STRING
    'Description' testDescription=STRING
    'Testmetrics'
    '{'(testMetrics+=TestMetric)+'}'

    'Testcases'
    '{'(testCases+=TestCase)+'}'

    'TestSuites'
    '{'(testSuite+=TestSuite)+'}'

    'Workflow'
    '{'workflow = Workflow '}'
;

TestMetric:
    'TestMetricID' name=ID
    '{'
    'TestMetricName' testMetricName=STRING
    'TestMetricModule' testMetricModule=STRING
    'Description' testMetricDescription=STRING
    '}'
;

TestSuite:
    'TestSuiteID' name=ID
    '{'
    'TestSuiteName' testSuiteName= STRING
    'NumberOfMaxIteration' iteration = Iteration
    'IntervalBetweenTests' '{' ('fixedInterval' fixedInterval=INT
                    | 'randomizedInterval' randInterval=Range
                    | 'sequenceFixedInterval' seqFixedInterval=ListInt)'}'
    'Offset' off=INT
    'Timeout' timeout = INT
    'BoundTestCases' '{' boundTestCases+=[TestCase]+ '}'
    '}'
;

Workflow:
    'WorkflowID' name=ID
    'WorkflowName' workflowName = STRING
    'WorkflowModule' workflowModule = STRING
    'BoundTestSuites'
    '{'
    boundTestSuites+=[TestSuite]+
    '}'
;

TestCase:
    'TestCaseID' name=ID '{'
    'TestCaseName' testCaseName=STRING
    'TestCaseModule' testCaseModule=STRING
    'Order' order=INT
    ('InputParameters' '{' inputParams+=Parameter '}' )?
    'AssertParameters' '{' (assertParams+=Parameter)+ '}'
    '}'
;
```

```
67
68  Iteration:
69          (count=INT | infinite='infinite')
70  ;
71
72  Parameter:
73          params += KeyValue (',' params+=KeyValue)*
74  ;
75
76  KeyValue:
77          '<' key = STRING ':'(intValue = INT
78                              | stringVal = STRING
79                              | listString = ListString
80                              | listInt = ListInt) '>'
81  ;
82
83  ListString:
84          '[' elements+=STRING ( ',' elements+=STRING)* ']'
85  ;
86
87  ListInt:
88          '[' elements+=INT ( ',' elements+=INT)* ']'
89  ;
90
91  Range:
92          '[' leftBound=INT ( ',' rightBound=INT) ']'
93  ;
94
95  @Override
96  terminal ID:
97          '^'?('a'..'z'|'A'..'Z'|'-')('a'..'z'|'A'..'Z'|'-'|'0'..'9')*
98  ;
```

Listing 6.4 shows a test definition example using the XText grammar definition of ConTest. Note that this example contains the identical information than Listing 6.2 which shows an example of a test definition using the EBNF grammar definition of ConTest. When comparing Listing 6.2 and 6.4, we can observe that the representation differs. The main reason for this is that the XText grammar supports cross-references which allows defining, e.g., test cases as separate blocks and later reference them using an ID.

Listing 6.4: Example of a continuous test definition using ConTest build with XText grammar

```
1
2  TestID d9ecdea3-e9da-4f78-b041-69b7509e3462
3  TestName "AvailabilityTestOfLoadBalancer"
4  Description "Continuously tests the availability of the load balancing
5              component of a cloud service"
6  Testmetrics {
7    TestMetricID YearlyAvailability {
8      TestMetricName "YearlyAvailabilityMetric"
9      TestMetricModule "de.fraunhofer.aisec.clouditor.metrics.YearlyAvailabilityMetric"
10     Description "Calculates the ratio between measured downtime since test
11                 start and 31557600 seconds in a year of 365.25 days"
12     }
13
14   TestMetricID DailyAvailability {
15     TestMetricName "DailyAvailabilityMetric"
16     TestMetricModule "de.fraunhofer.aisec.clouditor.metrics.DailyAvailabilityMetric"
17     Description "Calculates the daily availability starting from
18                       00:00 am to 23:59 (UTC)"
19   }
20 }
21
22 Testcases {
23   TestCaseID PingTest {
```

```
24        TestCaseName "PingTestCase"
25          TestCaseModule "de.fraunhofer.aisec.clouditor.testcases.PingTestCase"
26          Order 1
27          InputParameters {
28            <"count":10>,
29            <"host":"10.244.250.9">
30          }
31          AssertParameters {
32            <"round-trip-avg-$lte":"50 ms">,
33            <"round-trip-sd-$lte":"25 ms">
34          }
35    }
36    TestCaseID TCPTest {
37      TestCaseName "TCPTestCase"
38      TestCaseModule "de.fraunhofer.aisec.clouditor.testcases.TCPTestCase"
39      Order 1
40      InputParameters {
41        <"probe":10>,
42        <"host":"10.244.250.9">,
43        <"port":22>
44      }
45      AssertParameters {
46        <"response-time-avg-$lte":"75 ms">,
47        <"response-time-max-$lte":"100 ms">
48      }
49    }
50  }
51
52  TestSuites {
53    TestSuiteID PingTCPAvailabilityTestSuite {
54    TestSuiteName "AvailabilityTestSuite"
55    BoundTestCases {
56      PingTest
57      TCPTest
58      }
59    NumberOfMaxIteration 1000000
60    IntervalBetweenTests {
61      randomizedInterval [30,60]
62      }
63    Offset 15
64    Timeout 600
65    }
66  }
67
68  Workflow {
69    WorkflowID PingTCPAvailabilityWorkflow
70    WorkflowName "BasicIterationWorkflow"
71    WorkflowModule "de.fraunhofer.aisec.clouditor.workflow.BasicIterationWorkflow"
72    BoundTestSuites {
73      PingTCPAvailabilityTestSuite
74    }
75  }
```

Generating Clouditor configurations from ConTest As a unified configuration language, the implementation of ConTest only becomes meaningful if we can use this language to generate configurations for specific continuous test implementations. One example implementation of the framework introduced in Section 4.3 is called *Clouditor's engine* and is described in Chapter 4.4. Clouditor's engine was used to implement all of the example test scenarios presented in Chapter 5. Test configurations used by Clouditor's engine are written in YAML[110]. Thus we have to implement a code generator which translates the constructs of

[110] `http://yaml.org/spec/1.2/spec.html` [Accessed: 2018-12-13]

ConTest to YAML constructs used by Clouditor.

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.

Listing 6.5 shows an extract of the code generator's compile method[111] that we implemented to generate YAML files which are used to configure Clouditor's engine, the example implementation of our framework described in Section 4.4. The complete documentation of that implementation can be found in Appendix B. Listing 6.6 shows the YAML representation of the test definition which is generated from the test definition written in ConTest (see Listing 6.4).

Listing 6.5: Compile method of XText code generator to translate ConTest to YAML

```
1  def compile(Test ct) '''
2      «var length2 = "  "»
3      «var length4 = length2 + "  "»
4
5      name: «ct.testName»
6      id: «ct.name»
7      description: «ct.testDescription»
8
9      metrics:
10     «FOR m : ct.testMetrics»
11     «length2»- class: « m.testMetricModule»
12     «length4»name: « m.testMetricName»
13     «length4»description: « m.testMetricDescription»
14     «ENDFOR»
15
16     testCases:
17     «FOR tc : ct.testCases»
18     «length2»«tc.testCaseName»:
19     «length4»'@id': «tc.name»
20     «length4»'class': «tc.testCaseModule»
21     «length4»order: «tc.order»
22     «IF tc.inputParams !==null»
23         «FOR ip : tc.inputParams»
24             «FOR kv : ip.params»
25                 «length4»«kv.key»: «identifyParams(kv)»
26             «ENDFOR»
27         «ENDFOR»
28     «ENDIF»
29     «FOR ap : tc.assertParams»
30         «FOR kv : ap.params»
31             «length4»«kv.key»: «identifyParams(kv)»
32         «ENDFOR»
33     «ENDFOR»
34
35     «ENDFOR»
36     workflow:
37     «length2»class: «ct.workflow.workflowModule»
```

[111] Note that line breaks from lines 46 to 49 of Listing 6.5 are added for enhanced readability and have to be removed to correctly generate the YAML file shown in Listing 6.6, i.e., the code in line 47 – 49 has to be appended to line 46.

```
38      «length2»name: «ct.workflow.workflowName»
39      «length2»testSuites:
40      «FOR ts : ct.testSuites»
41      «length4»«ts.name»:
42      «length4»name: «ts.name»
43      «length4»label: «ts.testSuiteName»
44      «length4»randomized: «IF ts.randInterval !== null»true «ELSE»false«ENDIF»
45      «length4»iteration: «IF ts.iteration.infinite!==null»-1«ELSE»«ts.iteration.count»«ENDIF»
46      «length4»interval: «IF ts.randInterval !== null»
47      [«ts.randInterval.leftBound»,«ts.randInterval.rightBound»]«ELSE»
48      «IF ts.seqFixedInterval !==null»«ts.seqFixedInterval.elements»«ELSE»
49      [«ts.fixedInterval»]«ENDIF»«ENDIF»
50      «length4»offset: «ts.off»
51      «length4»timeout: «ts.timeout»
52      «length4»testCases: [«FOR tc:ct.testCases SEPARATOR ','»'@ref': «tc.name»«ENDFOR»]
53      «ENDFOR»
54  '''
```

Listing 6.6: YAML file generated from ConTest


```
1
2   name: AvailabilityTestOfLoadBalancer
3   id: d9ecdea3-e9da-4f78-b041-69b7509e3462
4   description: Continuously tests the availability of the load balancing
5                component of a cloud service
6
7   metrics:
8     - class: de.fraunhofer.aisec.clouditor.metrics.YearlyAvailabilityMetric
9       name: YearlyAvailabilityMetric
10      description: Calculates the ratio between measured downtime since test
11                   start and 31557600 seconds in a year of 365.25 days
12
13    - class: de.fraunhofer.aisec.clouditor.metrics.DailyAvailabilityMetric
14      name: DailyAvailabilityMetric
15      description: Calculates the daily availability starting from
16                   00:00 am to 23:59
17
18  testCases:
19    PingTestCase:
20      '@id': PingTest
21      class: de.fraunhofer.aisec.clouditor.testcases.PingTestCase
22      order: 1
23      count: 10
24      host: 10.244.250.9
25      round-trip-avg-$lte: 50 ms
26      round-trip-sd-$lte: 25 ms
27
28    TCPTestCase:
29      '@id': TCPTest
30      class: de.fraunhofer.aisec.clouditor.testcases.TCPTestCase
31      order: 1
32      probe: 10
33      host: 10.244.250.9
34      port: 22
35      response-time-avg-$lte:75 ms
36      response-time-sd-$lte:100 ms
37
38  workflow:
39    class: de.fraunhofer.aisec.clouditor.workflow.BasicIterationWorkflow
40    name: BasicIterationWorkflow
41    testSuites:
42      PingTCPAvailabilityTestSuite:
43          name: PingTCPAvailabilityTestSuite
44          label: AvailabilityTestSuite
45          randomized: true
46          iteration: 1000000
47          interval: [30,60]
```

```
48        offset: 15
49        timeout: 600
50        testCases: ['@ref': PingTest, '@ref': TCPTest ]
```

6.3 Summary and discussion

In this chapter, we introduced a domain-specific language (DSL) called *ConTest* to rigorously define continuous tests. To that end, we first presented an approach how to develop a DSL which was proposed by Mernik et al. [283]. Following this approach, we went through the following steps:

- *Motivate the necessity to build a DSL:* The purpose of ConTest is to provide a general, unified representation of tests which is agnostic to concrete implementations but adheres to the building blocks of our framework to support continuous test-based certification introduced in Chapter 4. This not only allows us to rigorously compare the evidence produced by different test implementations but also ensures conformance with our framework by having a developer provide a code generator with which test definitions written in ConTest can be translated into the target configuration language used by a specific test implementation. That way, ConTest serves as a unified configuration language for tests supporting continuous cloud certification.
- *Analysis of domain-specific constructs:* We investigated which parts of the building blocks of our framework are suitable to be used for a general representation of a continuous test, including test cases, test suites, workflow, test metrics and preconditions.
- *Design ConTest:* We defined ConTest using Extended Backus-Naur-Form (EBNF), a language to describe context-free grammars.
- *Implement ConTest:* In order to implement ConTest, we used the language development tool *XText* which uses a variant of EBNF. Furthermore, we implemented a code generator which translates the constructs of ConTest to YAML configuration files which are used to configure tests for Clouditor's engine, one example implementation of our framework (see Section 4.3) described in Section 4.4.

Our approach has the following limitations: Having a developer provide a code generator to translate from ConTest to an implementation-specific configuration language of a test only guarantees that the configuration of the test implementation adheres to the building blocks of our framework. A test which has been implemented by the developer may substantially deviate from our framework, for example, a test suite may also succeed even if not all test cases bound to the suite pass. Furthermore, the code generator provided by the developer may not utilize all constructs of ConTest to generate implementation-specific test definitions, thus rendering the generated configuration incomplete. Naturally, in these cases, rigorously comparing evidence produced by test implementations is not feasible because the tests' implementations deviate from the specification of our framework's building blocks. However, through using a formally defined DSL, tests can be defined rigorously where translation into a target configuration language using code generators has a crucial advantage: It makes the relationship between the building blocks of our framework and the implementation-specific configuration of a test implementation explicit, discernible and comprehensible.

Chapter 7

Evaluation of continuous test results

In the previous chapters, we introduced a framework to design and represent tests supporting continuous cloud services certification (Chapter 4 and 6) and demonstrated how this framework can be applied in example scenarios (Chapter 5). However, as pointed out in *Research Challenge 3: Accuracy and precision of continuous test results* described in Section 1.2.3, erroneous test results can decrease customers' trust in test results and can lead to providers disputing results of a continuous test. It is therefore vital to evaluate how accurate and precise produced tests' results are, that is, we have to answer the question: How close are particular test results to their true values?

Systematic errors are the reason why measured values, i.e., test results may not agree with true values. These errors can be induced by implementation and configuration errors of the measuring device, that is, any component used to implement a continuous test. Without comparing a test's results to the ground truth, it is hard to make a statement on how well a specific test works in producing evidence indicating a control's satisfaction or violation. Recall the example test scenarios presented in Chapter 5: The test designs implementing these scenarios leveraged external, existing tools such as Nmap, SQLMap or sslyze. It is obvious that any erroneous behavior these external tools may exhibit are inherited by the test using them which, in turn, can lead to systematic errors in the test results produced. Further inaccuracy may be caused by errors in the test environment or in the implementation of the test framework introduced in Chapter 4.

Addressing this challenge, this chapter presents a method which consists of intentionally manipulating cloud service properties to violate control satisfaction – thereby establishing the ground truth – and, on this basis, experimentally evaluate accuracy and precision of continuous test results. This method allows us to compare alternative test designs as well as alternative test configurations. Furthermore, it permits us to infer general conclusions about the accuracy of a specific continuous test. Parts of the contents of this chapter were published in [127] and [249].

First, we define the terms accuracy and precision in context of continuous tests of cloud services. Thereafter, we give a high-level overview of how our method works (Section 7.2) and describe how to intentionally violate cloud service properties leading to non-compliance of the service with a certificate's controls (Section 7.3). Then we introduce novel accuracy and precision measures to evaluate continuous tests (Section 7.4), including the inference of conclusions about the general accuracy of a test. Finally, we present experimental results of applying our method to evaluate and compare tests which aim to support certification of controls related to the properties availability and security (Section 7.5).

7.1 Accuracy and precision of continuous test results

In Section 4.3.5, we introduced test metrics which allow us to evaluate statements over cloud services properties. Test metrics reason about sequences of test results produced through repeatedly testing a cloud service's properties, thereby generating measurements. The question at this point is what errors these measurements may possess and how do these errors affect our conclusion about whether a cloud service satisfies a control or not.

In this section, we will define what *accuracy* (Section 7.1.1) and *precision* (Section 7.1.2) mean in the context of continuous test results. To that end, we draw on standard measurement theory and statistical methods which are used throughout various fields of experimental science. The basic definitions of concepts such as *accuracy* and *precision* used within this section are based on [293], [294] and [295]. Further, statistical methods described within this section are comprehensively covered in the literature, e.g., [296][297][298].

Note that we derive our definitions of accuracy and precision of continuous test results from the set of universal test metrics introduced as part of the framework in Chapter 4. These definitions are therefore only applicable to test designs implementing these test metrics. It is important to note that there exist numerous other possible test metric definitions which can be used to define accuracy and precision. Yet current research does not offer definitions of accuracy and precision in the context of test-based evidence production techniques (see Chapter 3 for a comprehensive overview of related work). Thus future work may propose accuracy and precision definitions derived from other test metrics and thus complement those presented hereafter.

7.1.1 Accuracy

When analyzing experimental data, the *accuracy* of the measurement describes whether the measured value agrees with the accepted value. This *accepted* or *true* value can be determined by previous observations or theoretical calculations. Therefore, the concept of accuracy may only be used to analyze experimental data if the goal is to compare the experimental results with known values.

Evaluating the accuracy of continuous test results depends on the test metrics which the continuous test employs. Recall the four universal test metrics brC, $fpsC$, $fpsD$ and $cpfsD$ introduced in Section 4.3.5. Accuracy in context of these test metrics can be outlined as follows:

- *Basic-Result-Counter (brC):* Counts the number of passed and failed tests. A basic test result is accurate if it indicates that a control is not satisfied by the cloud services at a time where the cloud service indeed does not comply with the control. Analogously, a basic test result is also accurate if it indicates satisfaction of a control by a cloud service at a time where the service indeed complies with the control.

- *Failed-Passed-Sequence-Counter ($fpsC$):* Counts the number of observed Failed-Passed-Sequences (fps). A fps is accurate if the cloud service actually does not comply with a control during the time indicated by the fps.

- *Failed-Passed-Sequence-Duration ($fpsD$):* Describes the time elapsed between the first failed test and the last passed test of a fps. Thus, the measured value of a $fpsD$ is accurate if it agrees with the actual duration of the temporary non-compliance of a cloud service.

- *Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$):* Describes the accumulated time during which a control is not satisfied. The measured value of $cfpsD$ is accurate if it matches the actual duration of the temporary non-compliance of cloud service within a specified interval.

Systematic errors are the reason why measured values may not agree with accepted values. These errors may occur because of, e.g., erroneous implementation and configuration of the measuring device. Identifying the causes of systematic errors is usually non-trivial. In our case, this measuring device consists of any component used to implement the framework to design tests to support continuous cloud services certification described in Chapter 4.

Systematic errors of continuous tests vary depending on the test metric. In Section 7.4, we will explore accuracy measures for each of the above test metrics which allow us to quantify the disagreement between measured values and true values. Furthermore, as we will detail in Section 7.3, we establish the true values through intentionally manipulating cloud services to *not* comply with controls which a test aims to validate. Thus, we know the true values and can compare them with the measured ones produced by the continuous test under evaluation, thereby allowing us to determine the accuracy of the test. However, the remaining problem is that the systematic error a test makes can vary due to *random errors*. This brings us to the concept of *precision* which we will explain in the next section.

7.1.2 Precision

Precision refers to the closeness of agreement between successively measured values which were carried out under identical conditions [294]. If we neglect systematic errors, then those repeatedly executed measurements result in a range of values which spread about the true value. The reason for this spread are *random errors*. These errors are caused by unknown and unforeseeable changes in the experiment, e.g., fluctuation in the network delay to due electronic noise. The smaller the random errors, the smaller the range of values, and thus the more precise the measurement [293]. Hence, we can say that the level of precision which experimental measurements can reach is constrained by random errors.

7.1.2.1 Arithmetic mean

Let's assume that we have observed some repeated measurements $X = \langle x_1, x_2, \ldots, x_n \rangle$ which only have random errors. The question now is: What is true value of these measurements? The answer – in statistical terms – is that we can use the values of sample distribution X to estimate the expected value μ of the parent distribution Y. The best estimate for μ which can be derived from these measurements is the arithmetic mean. Using the values of X, we then compute the sample mean

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

which serves as our estimate of μ. The intuition behind averaging is that random errors are equally likely to be above as well as below the true value. Thus, through averaging, we evenly divide the random error among all observations.

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

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

The reason why estimating the population mean μ and population proportion p based on $\bar{x}$ and $\bar{p}$ works is provided by the laws of large numbers: The *weak law of large numbers* states that if the number of samples n that we generate from the distribution Y goes to infinity, then the probability of making a random error larger than ϵ goes to zero:

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

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

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

This leads us to the following key insight: Both laws of large numbers suggest that if we generate a sufficiently large number of samples – i.e., take a sufficient large number of measurements – then we can produce an estimate $\bar{x}$ with a random error $\epsilon = |\bar{x}_n - \mu|$ which is as small as we desire. Put differently: With a sufficiently large number of measurements, our estimate converges to the true value plus systematic error. However, neither law tells us how many measurements we have to conduct in order to reduce ϵ below a particular threshold.

7.1.2.2 Standard deviation

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

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

The standard deviation takes all values of X into account and provides the average distance of a measurement value to the mean. Now, if we observe another measurement and want to know whether it is a common or exceptional value, then we can use sd. First, we standardize the newly observed value x by computing so-called *z-scores*:

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

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

The sd has one important drawback: Adding more measurement values to X increases the precision with which we estimate the population mean μ because it decreases the random error. However, when conducting more measurements, the standard deviation of X remains relatively stable. Therefore, the standard deviation is *not* a good measure to describe the error of the sample mean, that is, how close the sample mean is to the population mean.

7.1.2.3 Standard error

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

The calculation of the standard error depends on the selected statistic. The *se* for the sample mean $\bar{x}$ is computed as follows:

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

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

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

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

7.1.2.4 Confidence intervals

We now combine the notion of the standard error with the assumption that the sampling distribution approximately follows a normal distribution. This allows us to estimate the precision of the sample mean and the sample proportion by constructing confidence intervals for the sample mean and for the sample proportion. *Confidence intervals* are a special type of interval estimates which – in contrast to point estimation like $\bar{x}$ and $\bar{p}$ – give a range of probable values of an unknown parent's distribution parameter.

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

- *Confidence level (CL):* The fraction of all possible samples which are expected to include the true parameter of the unknown parent distribution. As an example, consider all possible samples X are drawn from the distribution of Y and for each a 99% confidence interval for the sample mean is computed. In this case, 99% of the computed confidence intervals include the population mean, i.e., the mean of the distribution of Y.
- *Statistic:* The property of a sample that is used to estimate population parameter's value. For our purposes, we use the sample mean ($\bar{x}$) and the sample proportion ($\bar{p}$).
- *Margin of error (E):* This constitutes the interval estimation by defining the range above and below the sample statistic. The calculation of E depends on the standard error which, in turn, depends on the selected statistic (see previous paragraph). For the sample mean $\bar{x}$, the margin of error is

$$E_{\bar{x}} = t_{CL} \times se_{\bar{x}}$$

where t_{CL} is the value that separates the middle the area of the t-Distribution according to the chosen confidence level CL, e.g., 95%, and the standard error of the mean $se_{\bar{x}}$.

For the sample proportion $\bar{p}$, the margin of error is

$$E_{\bar{p}} = z_{CL} \times se_{\bar{p}}$$

where z_{CL} is the z-value that separates the middle area of the standard normal distribution according to the selected confidence level CL, e.g., 99%, and the standard error for the proportion $se_{\bar{p}}$.

7.1.2.5 Calibrating precision

Recall that at the end of the section 7.1.2.1 on the arithmetic mean, we pointed out that the laws of large numbers justify making a point estimate of a parent's distribution parameter, e.g., using the sample mean $\bar{x}_n$ to estimate the mean μ of the distribution of Y. However, we do not know how close this estimate is to the true value (plus systematic error), that is, how large is the error ϵ for a given sample size n?

Having introduced how to construct confidence intervals for sample means and proportions, the sample size n can be used as a parameter to determine the number of samples which are needed for a desired margin of error $\hat{E}$. Thus, we can calibrate our experiment according to the desired precision. To that end, $E_{\bar{p}}$ and $E_{\bar{x}}$ are solved for the sample size n which gives us

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

and

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

In practice, one apparent problem of solving these formulas is that they have to be solved prior to executing the experiment to evaluate a continuous test. Thus, we may not have observed values to plug in for $\bar{p}$ and $sd_{\bar{x}}$. In this case, we have to make an educated guess, otherwise we can use historical values previously observed.

7.2 Overview of the evaluation process

The accuracy and precision of results produced by a specific continuous test depend on various factors, such as implementation of the test, test environment and usage of external tools. Without experimental evaluation, it is thus hard to make a statement on how well a specific test works in producing evidence indicating whether a control is satisfied or violated. The evaluation process presented hereafter aims to support such experimental evaluation of test results and thus provides comparability between alternative test designs.

Our approach treats the *continuous test under evaluation* as a black box. Therefore, no information about the internal composition and implementation of the test is needed, e.g., if and which external tools are used. We only observe results produced by the test during an experiment where we trigger violations of the control whose validation the tests seeks to support. Put differently: Correct results as well as errors of the test under evaluation follow some unknown distributions. We take samples from these unknown distributions by running

experiments during which we intentionally violate controls. Based on these experiment results, we infer conclusions about the general accuracy of the test.

Figure 7.1 provides a high-level overview of our method. As part of configuring control violations, we randomize duration of and time between each control violation event within some specified limits (Step 1). Then the test is configured according to the framework described in Chapter 4 (Step 2): Selecting test cases, setting test suites parameters and choosing a workflow. Thereafter, we start executing the control violation sequence and the continuous test at the same time (Step 3) and observe whether violations are detected by the test (Step 4). Provided our sample size is sufficiently large, i.e., the test has produced enough results, we infer parameters of the unknown parent distribution, that is, we draw conclusions about the general accuracy of the test under evaluation (Step 5). These inferences are considered valid with regard to the test and control violation configuration parameters.

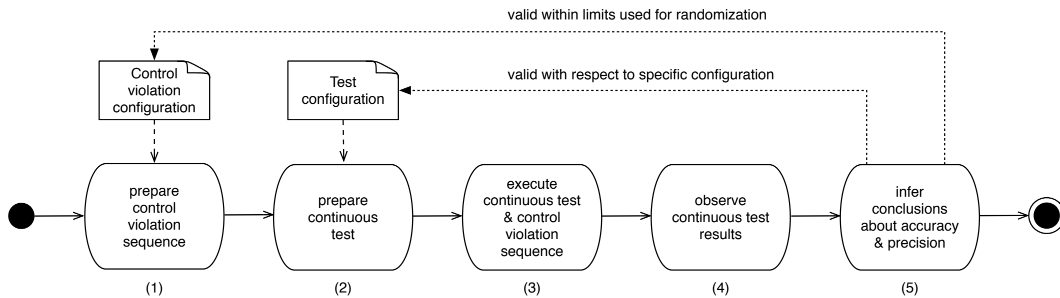


Figure 7.1: Experimental evaluation of the accuracy and precision of continuous test results

7.3 Control violations

A *control violation* has the purpose to manipulate one or more properties of a cloud service under test to mock violations of controls which a specific continuous test aims to validate. Therefore, control violations establish the ground truth, that is, the accepted values to which results produced by a test are compared.

7.3.1 Control violation sequence

We are aiming at evaluating the accuracy and precision of test results produced by a continuous test. Recall that one of the key drivers for continuously testing cloud services is founded on the assumption that a cloud service's property may change over time where these changes can lead to control violations. As a consequence, the properties of a cloud service may comply with controls at some time while at other times, they do not.

In order to mimic this non-stationary behavior of cloud services' properties, control violations have to repeatedly create control violation events (cve) over time. During a cve, a cloud service's properties are manipulated so that the service does not comply with the control. Between two successive cve, the cloud service's properties satisfy the control. We describe a control violation as the sequence

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

As Figure 7.2 shows, each cve starts at cve^s and ends cve^e, thus having a duration of

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

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

$$cveW = cve^e_{i-1} - cve^s_i.$$

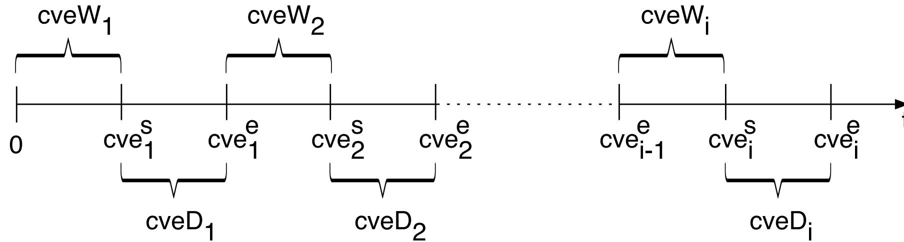


Figure 7.2: Sequence of control violation events cve

7.3.2 Control violation design

The design of a control violation is driven by the specific control which the continuous test under evaluation aims to validate. Thus, the question at this point is: Which properties of a cloud service have to be modified in order to violate a particular control?

A key insight at this point is that we do *not* aim to design a particular control violation which is *complete*, that is, which manipulate a cloud service in *any* possible way such that a particular control would be dissatisfied. Naturally, having a complete control violation design would be helpful to evaluate the completeness of the corresponding continuous test. Yet, if we were to design a complete control violation, then we would face a challenge similar to deriving executable tests from high-level, ambiguous control descriptions: We would have to interpret what it means for the control to be satisfied or dissatisfied on the implementation level of a cloud service instance. The difference to deriving executable tests is, however, that we were to design mechanisms which intentionally alter a cloud service's properties to violate the control.

However, the goal of our evaluation is *correctness* of a continuous test, that is, we want to evaluate how accurate and precise the results of the test under evaluation are. Therefore, we use the definition of the test under evaluation as a starting point to derive the design of the control violation. The two main steps in the control violation design process are:

1. *Inspect assert parameters:* The first step when designing a control violation consists of inspecting the definition of the test under evaluation. Recall that a single test result of a continuous test is produced by executing a test suite which fails if any test case bound the test suite fail. Thus, we have to inspect the assert parameters which are used to configure the expected outcome of each test case. Based on the assert parameters and on their configured value, we then derive which property of the cloud service has to be manipulated in order for these asserts to be dissatisfied.

   Consider, as an example, that a continuous test probes a set of ports to check whether the cloud service exposes sensitive interfaces. The assert parameters of the test definition will give us the ports which are considered sensitive, that is, should not be reachable. Manipulating the service such that is exposes some blacklisted ports depicts one example of designing control violation events.

2. *Specify control violation events:* The second step consists of deciding on the lower ($cveW^L$) and upper ($cveW^R$) limit of the duration between two successive control violation events $cveW$. Furthermore, the lower ($cveD^L$) and upper ($cveD^R$) limit of the time during which a cloud service's property is manipulated rendering it non-compliant have to be defined. In the following section, we will explain the purpose of randomizing duration of and interval between control violation events. Note that deciding on how many control violation events to trigger is driven by the selected precision measures which will be explained in detail in Section 7.4.

7.3.3 Standardizing control violation events

Control violations establish the ground truth against which we evaluate a specific test. To infer conclusions about the general accuracy of a test, ideally any possible sequence of any possible control violation event has to be experimentally evaluated. Naturally, this is infeasible in practice and we have to select a sequence of control violation events V which meets our time and space constraints. But how do we select a sequence V still allowing us to draw conclusions about the general correctness of a continuous test?

The answer consists of two parts: At first, we need to standardize the control violation event: For each cve we use to construct V, the duration of the control violation $cveD$ and the waiting time before start $cveW$ are selected randomly from intervals $[cveD^L, cveD^R]$ and $[cveW^L, cveW^R]$, respectively. Choosing these intervals' limits lets us configure a control violation sequence according to our space and time limitations. Secondly, we need to decide how many cve, i.e., $|V|$ are required to infer conclusions about the general accuracy of the continuous test under evaluation. This depends on the statistical inference method which, in turn, depends on the precision measure. We will address this question for each precision measure in Section 7.4.

7.4 Accuracy and precision measures

After having discussed how to intentionally violate controls for experimental purposes, this section describes models to estimate the accuracy and precision of continuous tests under evaluation. Hereafter, we refer to these models as *accuracy measures* and *precision measures*. These measures are based on the universal test metrics brC, $fpsC$, $fpsD$, and $cfpsD$ which were introduced in Section 4.3.5. In order to derive the accuracy and precision measures, each of the next four sections (7.4.1 – 7.4.4) follows these three steps:

1. *Evaluate test results:* We use the results produced by a continuous test during a control violation sequence and evaluate whether they are correct or erroneous. In the latter case, we specify the type of observed error which depends on the test metric, e.g., a false negative basic test result incorrectly suggests that a cloud service does not satisfies a control.

2. *Derive accuracy measures:* Using the evaluation of the test results as input, the accuracy measures then estimate if and how the measured values of test under evaluation deviate from the accepted, i.e., true values as established by the control violation sequence.

3. *Derive precision measures:* Based on the evaluation measures, the precision measures estimate how the measured values spread about the accepted value.

Note the set of accuracy and precision measures proposed in this section is not complete. Since current research has not yet proposed other approaches permitting to evaluate accuracy and precision of test-based evidence production techniques (a comprehensive overview of related work can be found in Chapter 3), it is therefore left to future work to build on the set of evaluation measures proposed hereafter and extend this set as needed.

7.4.1 Basic-Result-Counter

In this section, we describe how to estimate accuracy and precision of a continuous test using the Basic-Result-Counter test metric (*brC*). To that end, the next section describes the evaluation of test results using six evaluation measures. Thereafter, we detail how to use these evaluation measures to compute accuracy and precision measures (Section 7.4.1.2 and 7.4.1.3).

7.4.1.1 Evaluation of test results

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

True negative basic test result counter (brC^{TN}) A test produces a true negative result if the test fails at a time when a control is violated. More specifically, as shown in Figure 7.3, a br^{TN} is produced if a failing test starts (tsr^s) after a control violation event starts (cve^s) and the test ends (tsr^e) before the event ends (cve^e):

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

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

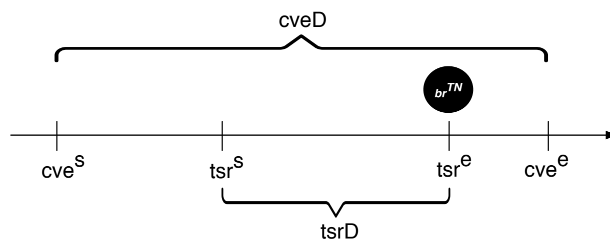


Figure 7.3: True negative basic test result (br^{TN})

True positive basic test result counter (brC^{TP}) A true positive test result is produced if the test passes at a time when *no* control is violated. As shown in Figure 7.4, a passing test producing a true positive result starts after the previous control violation event ends and ends before the next control violation event starts:

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

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

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

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

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

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

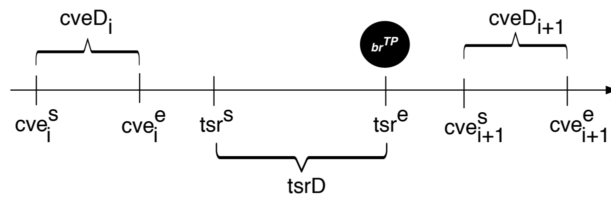


Figure 7.4: True positive basic test result (br^{TP})

False negative basic test result counter (*brC*^{FN}) If a test fails at a time when *no* control is violated, then the test produces a false negative test result. When comparing Figure 7.4 and Figure 7.5, it becomes evident that the definition of a false negative test result is analogous to the definition of a true positive test result. The only difference being that the test result incorrectly fails, indicating a control violation:

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

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

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

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

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

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

False positive basic test result counter (*brC*^{FP}) If a test passes at a time when a control is violated, then the incorrectly passing test produces a false positive result (br^{FP}). The definition of br^{FP} is similar to a true negative result (see Figure 7.6), only that the test incorrectly passes, that is,

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

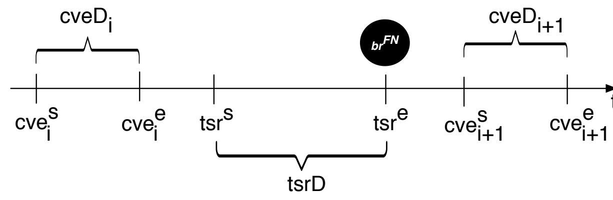


Figure 7.5: False negative basic test result (br^{FN})

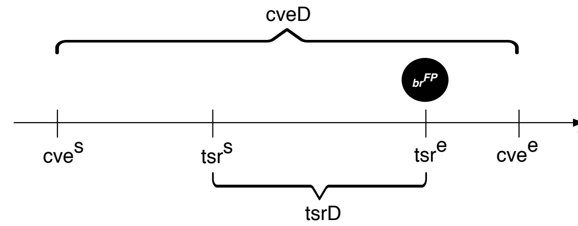


Figure 7.6: False positive basic test result (br^{FP})

Moreover, there is one special case: As shown in Figure 7.7, a passing test may cover one or more control violation events completely, that is,

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

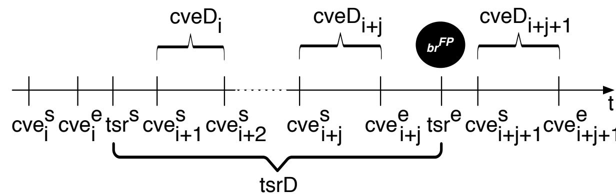


Figure 7.7: False positive basic test result (br^{FP}) covering multiple cve

Counting all false positive results is described by brC^{FP}.

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

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

ends (tsr^e) after the violation events starts (cve^s) and before the control violation ends (cve^e):

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

As an example, consider the following scenario: A test starts measuring available bandwidth of a virtual machine. Only after the test started, the bandwidth of the virtual machine is limited. Therefore, while at the beginning of the test no control was violated, later during the test it was. If the test in total determines that the available bandwidth was insufficient, then the test fails, producing a pseudo true negative result br^{PTN}.

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

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

While Figure 7.8 shows a br^{PTN} where a correctly failing test ends during a control violation event, Figure 7.9 depicts the case where a correctly failing test starts during a control violation event. In Figure 7.8, note the dotted line between the start of the test (tsr^s) and the start of the violation event (cve^s). It indicates that a test can cover multiple control violation events. Similarly, in Figure 7.9, the dotted line between the end of the control violation event (cve^e) and the end of the test (tsr^e) illustrates that the test may cover multiple control violation events.

If a test covers multiple cve, then this implies that a test takes longer to complete ($tsrD$) than the duration of the control violation event ($cveD_i$), that is, $tsrD > cveD_i$. This leads to a more general insight: A correctly failing test which completes at $[cve_i^s, cve_i^e]$ and for which $tsrD > cveD_i$ holds, will always produce a pseudo true negative test result.

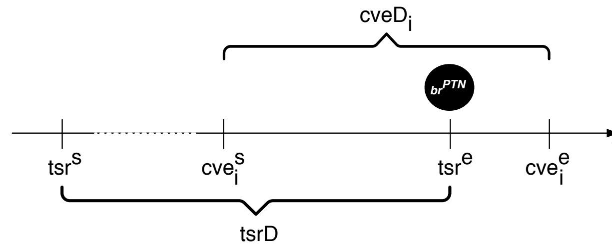


Figure 7.8: Pseudo true negative basic test result (br^{PTN}) where the test ends during cve

Lastly, we use brC^{PTN} to count any occurrence of pseudo true negative test results.

Pseudo false positive basic test result counter (brC^{PFP}) A test produces a pseudo false positive result if the test partially overlaps with a control violation event but incorrectly passes. Thus, the definition of br^{PFP} is similar to br^{PTN} presented in the previous paragraph, the only difference being that the test result is positive. As in the case of a br^{PTN}, a br^{PFP} can either end during a control violation event or it can start during a control violation event. Furthermore, a br^{PFP} may cover multiple control violation events. We count the occurrence of pseudo false positive results using brC^{PFP}.

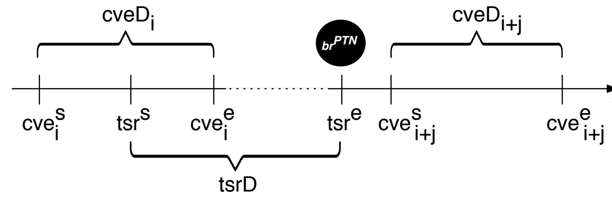


Figure 7.9: Pseudo true negative basic test result (br^{PTN}) where the test starts during *cve*

7.4.1.2 Accuracy measures based on *brC*

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

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

As a next step, we use these evaluation measures as input to compute accuracy measures. To that end, we draw on standard accuracy measures used in binary classification described by, e.g., [299], [300] and [301]. Hereafter, we describe which specific measures we selected and how to interpret them to evaluate the accuracy of a test under evaluation.

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

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

- *True negative rate (tnr):* This measure describes the proportion of correctly failed tests ($brC^{TN} + brC^{PTN}$) out of any test that should actually have failed ($brC^{TN} + brC^{PTN} + brC^{FP} + brC^{PFP}$). Using *tnr*, we can analyze the ability of a continuous test to correctly detect whether a cloud service complies with a control or not:

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

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

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

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

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

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

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

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

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

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

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

- *False omission rate (for):* This measure calculates the ratio between incorrectly failed tests (brC^{FN}) and all tests which failed ($brC^{TN} + brC^{PTN} + brC^{FN}$). Thus we can describe the proportion of test results produced by a continuous test that should have passed within all produced test results that failed:

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

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

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

7.4.1.3 Precision measures based on *brC*

All the accuracy measures which are based on evaluating basic test results (br), e.g., true negative rate (tnr), false positive rate (fpr), and negative predictive value (npv) have in common that they are *proportions*, that is, they give us the fraction of, e.g., correct test results out of any observed test results. This leads to the key idea at this point which is to construct confidence intervals for these proportions, that is, estimate the precision of these accuracy measures using interval estimates.

Consider, as an example, computing a confidence interval of 95% for npv^{brC}. This interval estimate allows us to state that we are 95% confident that the npv^{brC} of a continuous test under evaluation is contained in the interval. This inference is valid with respect to the definition of the test and the control violation configuration.

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

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

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

$$se_{npv} = \sqrt{\widehat{npv}^{brC} \times \frac{(1 - \widehat{npv}^{brC})}{n}}.$$

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

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

As stated above, we want to use the standard normal distribution to look up the value for $z_{95\%}$. This requires the sampling distribution of the proportion to be Gaussian. In order to find the required sample size $\tilde{n}$, we solve the margin of error $E^{95\%}_{npv} = z_{95\%} \times se$ for the sample size $\tilde{n}$:

$$\tilde{n} = \frac{z_{95\%} \times \widehat{npv}^{brC} \times (1 - \widehat{npv}^{brC})}{\hat{E}^2} \tag{7.1}$$

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

Recall that in Section 7.3.2 and 7.3.3, we posed the question of how many control violation events $|V|$ are needed in order to allow for inferring conclusions about the general accuracy of the continuous test under evaluation. Continuing our example for npv^{brC}, finding the required size of V can be formulated as an optimization problem:

$$\begin{aligned} \text{minimize} \quad & |V| \\ \text{subject to} \quad & \tilde{n} \leq brC^{TN} + brC^{PTN} + brC^{FN} \end{aligned} \tag{7.2}$$

We have to trigger at least as many control violation events cve as are required to observe $\tilde{n}$ test results. Following the above steps, interval estimates for the remaining accuracy measures, that is, oac^{brC}, tnr^{brC}, tpr^{brC}, fnr^{brC}, fpr^{brC}, fdr^{brC}, ppv^{brC}, and for^{brC} introduced in Section 7.4.1 can be computed analogously.

7.4.2 Failed-Passed-Sequence-Counter

This section describes how to estimate the accuracy and precision of a continuous test under evaluation using the Failed-Passed-Sequence-Counter metric ($fpsC$). To that end, the next section describes the evaluation of test results using three evaluation measures. Thereafter, we detail how to use these evaluation measures to compute accuracy and precision measures (Section 7.4.2.2 and 7.4.2.3).

7.4.2.1 Evaluation of test results

In this section, we explain how we evaluate a continuous test based on the Failed-Passed-Sequence-Counter metric ($fpsC$). Recall that $fpsC$ counts the occurrence of Failed-Passed-Sequences (fps) which is a special sequence of test results, starting with a failed test and ending with the next passing test (see Section 4.3.5). A fps aims at detecting temporal violations of a control, that is, control violation events that persist for some time. In order to evaluate the results of a continuous test, we check if and how any fps overlaps with control violation events cve.

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

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

Note that a fps^{TN} can cover multiple cve. Figure 7.10 shows an example of a true negative fps whose first failed test produced a pseudo true negative result (br^{PTN}) which starts at tsr^s_j. This fps^{TN} covers two control violation events, that is, cve_i and cve_{i+1}. We use $fpsC^{TN}$ to count the number of fps^{TN} which were observed during a control violation sequence.

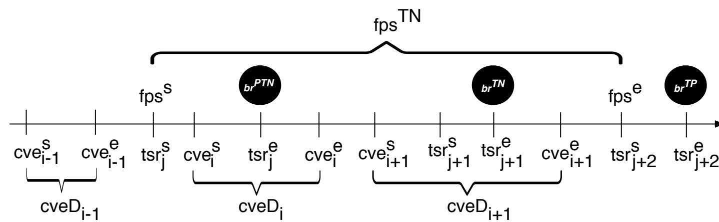


Figure 7.10: True negative Failed-Passed-Sequence (fps^{TN})

Furthermore, note that a true negative fps which detects the first control violation event during experimental evaluation is a special case: If no previous cve exists, then the following,

simplified definition of fps^{TN} is applied:

$$fps^{TN} = fps^s \leq cve_i^e \wedge cve_i^e < fps^e.$$

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

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

Figure 7.11 shows this basic version of a fps^{FN}. $fpsC^{FN}$ counts all occurrence of fps^{FN} observed during a control violation sequence. Yet false negative fps can also contain true

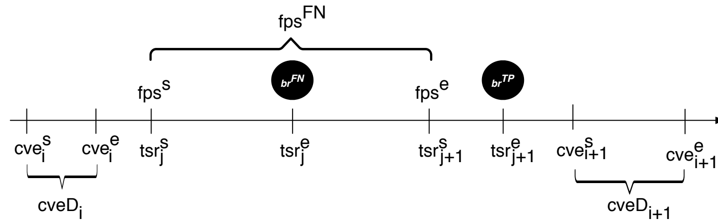


Figure 7.11: False negative fps

negative basic test results. This is the case if after a cve ended and before the next cve starts, that is, no control is violated, basic results still incorrectly indicate a control violation. Figure 7.12 shows an example of this error: After the control violation event cve_i ended at cve_i^e and before the next cve starts at cve_{i+1}^s, the test tsr_{j+1} produces a false negative test result at tsr_{j+1}^e.

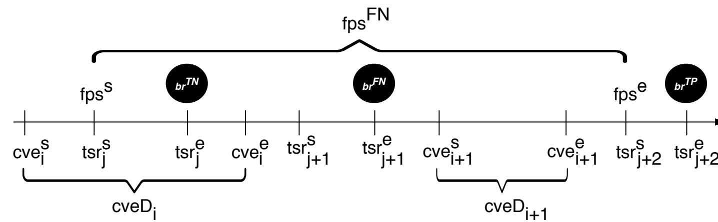


Figure 7.12: False negative Failed-Passed-Sequence (fps^{FN}) with true negative and false negative basic test result (br^{TN} & br^{FN})

Complementary indicators of the presence of fps^{FN} are the false omission rate (for^{brC}) and negative predictive value (npv^{brC}). These accuracy measures are calculated based on basic test results (see Section 7.4.1.2). The more incorrect negative basic test results are observed during evaluation of a continuous test, the higher for^{brC} and the lower npv^{brC}.

Lastly, the last test of a fps^{FN} can be a false positive, i.e., the last test result incorrectly suggests that the cloud service satisfies a control. Figure 7.13 shows one example of this error: After a test correctly failed at tsr_{j+1}^e, the next test incorrectly passes while the control is still violated, thereby producing a false positive test result (br^{FP}) at tsr_{j+2}^e. As a complementary

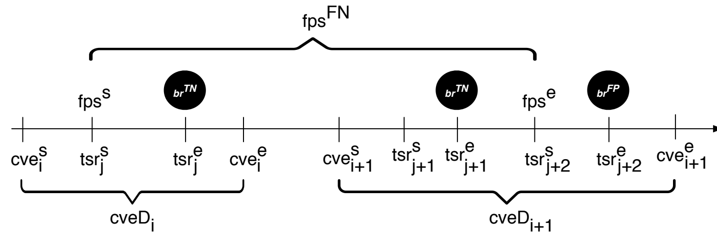


Figure 7.13: False negative Failed-Passed-Sequence (fps^{FN}) with false positive basic test result (br^{FP})

means to analyze this type of error, we can make use of the positive predictive value (ppv^{brC}) and false discovery rate (fdr^{brC}) introduced in Section 7.4.1.2: The more incorrect positive basic test results are observed, the higher fdr^{brC} and the lower ppv^{brC}.

False positive fps (fps^{FP}) A fps indicates that a cloud service does not satisfy a control over time. Therefore, if a control violation event is *not* detected by a continuous test, then this is considered a false positive fps. Figure 7.14 illustrates a cve that starts after the last fps ended (fps_j^e) and ends before the next fps starts (fps_{j+1}^s):

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

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

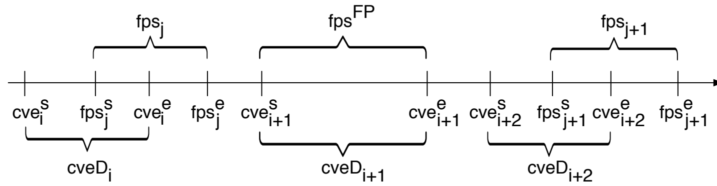


Figure 7.14: False positive fps

7.4.2.2 Accuracy measures based on $fpsC$

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

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

Next, we are going to use these evaluation measures to calculate accuracy measures. Analogous to the accuracy measures based on brC introduced in Section 7.4.1.2, we draw on standard measures used in binary classification. The following paragraphs explain which measures we selected and how these measures can be used to interpret the accuracy of a continuous test under evaluation to identify temporal violations of controls.

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

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

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

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

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

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

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

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

Using for^{fpsC}, we can make statements about how often a continuous test incorrectly suggested that a cloud service did not comply with a control for some time out of all detected control violation events.

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

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

On the basis of npv^{fpsC}, we can evaluate how many times a continuous test correctly indicated a control violation event out of all control violation events that the continuous test's results suggested.

7.4.2.3 Precision measures based on $fpsC$

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

There is one important difference: When determining the required number of control violation events $|V|$, we now have to trigger at least as many control violation events $|V|$ as are needed to observe $\tilde{n}$ fps. As an example, consider we want to construct a confidence interval for tnr^{fpsC}. The sample size n for tnr^{fpsC} consists of any control violation event that should have been detected, that is,

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

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

$$\begin{aligned} &\text{minimize} \quad |V| \\ &\text{subject to} \quad \tilde{n} \leq fpsC^{TN} + fpsC^{FP} \end{aligned} \tag{7.3}$$

Following the approach described above, the precision estimates for the remaining three accuracy measures fpr^{fpsC}, for^{fpsC} and npv^{fpsC} introduced in the previous paragraph can be computed analogously.

7.4.3 Failed-Passed-Sequence-Duration

This section describes how to estimate the accuracy and precision of a continuous test based on results produced by the Failed-Passed-Sequence-Duration test metric ($fpsD$). We begin with describing different types of errors that an $fpsD$ can make when attempting to determine the duration, start and end of a control violation event. Thereafter, we explain how we use these evaluation measures to estimate the accuracy and precision based on $fpsD$ (Section 7.4.3.2 and 7.4.3.3).

7.4.3.1 Evaluation of test results

The following paragraphs describe how to evaluate a continuous test based on results produced by the Failed-Passed-Sequence-Duration test metric ($fpsD$). Recall that $fpsD$ captures the time (e.g., in milliseconds) between the start of the first failed test (fps^s), i.e., first element of a fps, and the start of the next subsequent passed test (fps^e), i.e., last element of a fps (see Section 4.3.5).

Duration error of true negative fpsD ($efpsD^{TN}$) Having observed a true negative fps, we compute the difference between the duration of the fps, i.e., $fpsD = fps^e - fps^s$ and the duration $cveD$ of any control violation events which is covered by the fps. Figure 7.15 shows that a fps^{TN} can cover multiple cve. Yet it can, at most, cover all cve contained in the control violation sequence V:

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

Note that we do *not* calculate the absolute difference between $fpsD$ and covered $cveD$. This allows us to evaluate whether a $fpsD$ overestimates or underestimates the duration of a control violation event: If $efpsD^{TN} > 0$, then the $fpsD$ overestimates the duration of covered control violation events (Figure 7.15). Furthermore, if $efpsD^{TN} < 0$, the $fpsD$ underestimates the duration of covered control violation events (Figure 7.16). Naturally, if

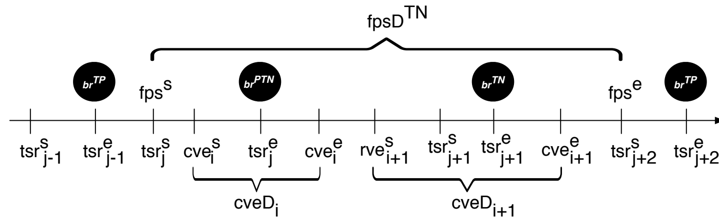


Figure 7.15: True negative Failed-Passed-Sequence-Duration ($fpsD^{TN}$) which overestimates total duration of cve_i and cve_{i+1}

$efpsD^{TN} = 0$, the $fpsD$ and the duration of the covered control violation events coincide perfectly.

Furthermore, we calculate the relative error that a fps makes when estimating the duration of i covered control violation events. To that end, we compute

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

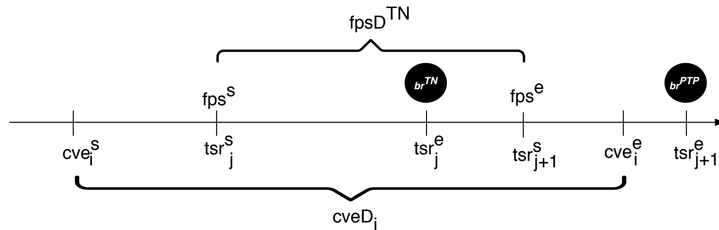


Figure 7.16: True negative Failed-Passed-Sequence-Duration ($fpsD^{TN}$) which underestimates duration of cve_i

Pre-duration error of true negative fpsD ($efpsD^{TN}_{pre}$) Up to this point, our error definition focused on the estimated duration of control violation events which is provided by a true negative $fpsD$. However, as Figure 7.17 illustrates, the start of a $fpsD^{TN}$ which estimates the start of the control violation event can be inaccurate, that is, $cve^s < fps^s$. Capturing this error, we compute the difference between the start of a fps, i.e., the start of the first failed test which detected a control violation event (fps^s), and the start of the control violation event (cve^s):

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

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

Post-duration error on true negative fpsD ($efpsD^{TN}_{post}$) Recall that the last basic test result of a true negative fps is always a true positive basic test result. Therefore, a fps^{TN} by definition only ends after the control violation event ends. Figure 7.17 shows $efpsD^{TN}_{post}$,

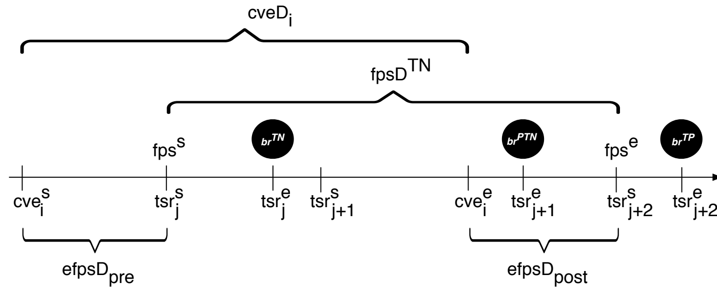


Figure 7.17: True negative Failed-Passed-Sequence-Duration ($fpsD^{TN}$) with $efpsD^{TN}_{pre} > 0$ and $efpsD^{TN}_{post} > 0$

that is, the resulting error which the last test result of a $fpsD^{TN}$ makes when determining the end of a control violation event. In order to describe this error, we compute the difference between the end of a control violation event (cve^e) and the end of the fps, that is, the start of the last test which passed:

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

Duration error of false negative fpsD ($efpsD^{FN}$) If a false negative fps is observed, then we consider the entire duration of that fps to be erroneous since it incorrectly indicates a duration of a control violation event. Figure 7.18 shows a $fpsD^{FN}$, it is defined as follows:

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

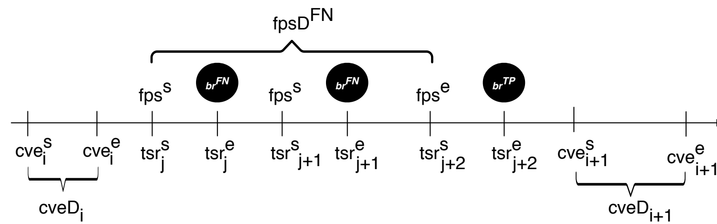


Figure 7.18: False negative Failed-Passed-Sequence-Duration ($fpsD^{FN}$)

Duration error of false positive fpsD ($efpsD^{FP}$) If we observe that a control violation event is not detected by a fps at all, then we treat this missed cve as a false positive fps. Consequently, the duration of a false positive fps is simply the duration of the missed control violation event (see Figure 7.19):

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

7.4.3.2 Accuracy measures based on $efpsD$

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

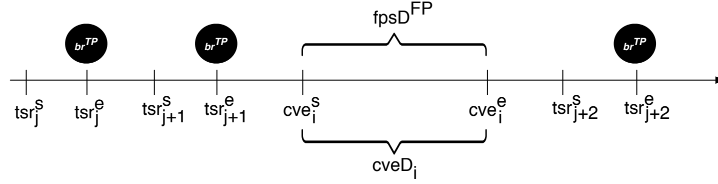


Figure 7.19: False positive Failed-Passed-Sequence-Duration ($fpsD^{FP}$)

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

When evaluating a continuous test, we may observe instances of any of the above errors. We treat observations of each type of error on $fpsD$ as separate distributions. Therefore, after having run a control violation sequence and the continuous test under evaluation, we expect to obtain at most five distributions. Note that, in practice, a continuous test may not produce any incorrect test results, i.e., neither br^{FN} nor br^{FP}. In these cases, we will not observe any instances of $efpsD^{FN}$. Further, a continuous test may not miss any control violation events, thus no $efpsD^{FP}$ is produced either. However, a continuous test *not* making any error on estimating the total duration, the start and the end of any control violation event is rather unlikely. The reason for this is that not observing any instance of $efpsD^{TN}$, $efpsD^{TN}_{pre}$, or $efpsD^{TN}_{post}$ requires the continuous test to always perfectly estimate duration, start and end of any control violation event. Therefore, we can expect to observe at least three distributions after having evaluated a continuous test's results based on $fpsD$, i.e., $efpsD^{TN}$, $efpsD^{TN}_{pre}$ and $efpsD^{TN}_{post}$.

In order to estimate the accuracy of a continuous test when measuring temporal violations of controls (e.g., in milliseconds), we compute the arithmetic mean ($\bar{x}$) for each of the observed distributions. $\bar{x}$ sums the values of a type of error and divides the result by the number of instances of that error type observed during evaluation of the continuous test. For example, in order to calculate the arithmetic mean for $efpsD^{TN}$, we add any instances i of $efpsD^{TN}$ contained in the sequence $EFPSD^{TN}$ and divide by the number of elements in $EFPSD^{TN}$:

$$\bar{x}^{TN} = \frac{(efpsD^{TN}_1 + efpsD^{TN}_2 + \cdots + efpsD^{TN}_i)}{|EFPSD^{TN}|}$$

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

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

7.4.3.3 Precision measures based on $efpsD$

Describing the precision of a continuous test under evaluation, we compute the following statistics:

- *Standard deviation (sd):* Measures the dispersion of values within a distribution. Continuing our example from the previous paragraph, the standard deviation of the values in $EFPSD^{TN}$ tells us how far values spread around its mean:

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

  Using sd, we can describe the variation of the different types of error which a continuous test makes when measuring the duration of control violation events. Furthermore, we will also use sd to compute the standard error of the mean which is needed to calculate confidence intervals which is explained in the following paragraph.

- *Confidence Interval for the sample mean:* We presented five types of errors a $fpsD$ may make when measuring the duration of a control violation event, e.g., $efpsD^{TN}$ and $efpsD^{FP}$. For each of these error types, we compute the mean $\bar{x}$ of the observed distribution as an accuracy measure. In order to make a statement about the precision of a continuous test, the core idea here is to construct a confidence interval for each mean.

  As an example, consider $efpsD^{TN}$, i.e., the mean error that a continuous test makes when determining the duration of a control violation event: A confidence interval on this mean allows statements such as we are 99% confident that the average error of a continuous test – with respect to the test defined and control violation configuration – makes when estimating the duration of a control violation event is contained in the interval. We compute this estimate with

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

  where $t_{99\%}$ is the value that separates the middle 99% of the area under the t-Distribution and se is the standard error which can be estimated with

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

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

$$\tilde{n} = \frac{\sigma^2 \times t_{99\%}^2}{\hat{E}^2} \tag{7.4}$$

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

  Inferring statements about the general accuracy of a continuous test based on the mean of, e.g., $\bar{x}_{TN}$ requires triggering a minimum number of control violation events. In

our example for $efpsD^{TN}$, we find the minimum size of V by solving the following optimization problem:

$$\begin{aligned} &\text{minimize} && |V| \\ &\text{subject to} && \tilde{n} \leq fpsD^{TN}. \end{aligned} \tag{7.5}$$

We need to trigger at least as many control violation events as are needed to observe $\tilde{n}$ $fpsD^{TN}$. Using these steps, interval estimates for the means of $efpsD^{TN}_{pre}$, $efpsD^{TN}_{post}$, $efpsD^{FN}$, and $efpsD^{FP}$ can be computed analogously.

As a complementary measure, we also compute the minimum and maximum (min & max), that is, the smallest and largest value for any type of error that was observed during evaluation of a continuous test. Using these statistics, we can describe the most extreme errors that a continuous test makes when measuring duration of control violation events. Furthermore, comparing min and max to the standard deviation can help to identify whether the test results produced during evaluation contain outliers.

7.4.4 Cumulative-Failed-Passed-Sequence-Duration

This section describes how to determine the accuracy of a continuous test based on the Cumulative-Failed-Passed-Sequence-Duration test metric ($cfpsD$). Hereafter, we first introduce three evaluation measures $cfpsD^{TN}$, $cfpsD^{FN}$, and $cfpsD^{FP}$ which are computed based on $fpsD^{TN}$, $fpsD^{FN}$, and $fpsD^{FP}$, respectively. Thereafter, we explain how these evaluation measures are used to estimate the accuracy of a continuous test under evaluation.

7.4.4.1 Evaluation of test results

This section describes how, based on the Cumulative-Failed-Passed-Sequence-Duration test metric ($cfpsD$), a continuous test can be evaluated. Recall that this metric accumulates the value of any $fpsD$ (e.g., in milliseconds) observed within a specified period of time, thus allowing to determine whether a cloud service satisfies a control with temporal constraints within that period (see Section 4.3.5).

True negative cfpsD ($cfpsD^{TN}$) Each value of a true negative $fpsD$ observed during evaluation of the continuous test is added, i.e.,

$$cfpsD^{TN} = fpsD^{TN}_1 + fpsD^{TN}_2 + \cdots + fpsD^{TN}_i.$$

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

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

$$cfpsD^{FN} = fpsD^{FN}_1 + fpsD^{FN}_2 + \cdots + fpsD^{FN}_i.$$

We can use $cfpsD^{FN}$ to capture the total measured duration of control violation events which the continuous test incorrectly indicated.

False positive cfpsD ($cfpsD^{FP}$) The sum of any false positive $fpsD$ which was produced by the continuous test under evaluation is computed by this measure, that is,

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

Using $cfpsD^{FP}$, we can describe the total duration of control violation events that were *not* detected by the continuous test under evaluation.

7.4.4.2 Accuracy measures based on $cfpsD$

The previous three paragraphs introduced three evaluation measures:

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

In order to determine the overall accuracy of a continuous test within a predefined period of time, that is, from start to end of the control violation sequence, we introduce the following three accuracy measures:

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

  $$ecfpsD^{TN} = cfpsD^{TN} - \sum_{i=1}^{|V|} cveD_i.$$

  The accuracy measure $ecfpsD^{TN}$ allows us to describe whether a continuous test overestimates or underestimate the accumulated duration of control violations within a specified period of time. If the continuous test overestimates the duration of the control violation events, then $ecfpsD^{TN} > 0$. Otherwise, if the continuous test underestimates the duration of the control violation events, then $ecfpsD^{TN} < 0$. Lastly, if $ecfpsD^{TN} = 0$, then the duration measured by the continuous test perfectly corresponds to the total duration of control violation events.

  Furthermore, we compute the ratio between $ecfpsD^{TN}$ and the total duration of all i control violation events contained in V, that is,

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

  Using $ecfpsD_{rel}^{TN}$, we can describe the relative measurement error that a continuous test makes when estimating the accumulated time during which a cloud service does not comply with a control.

- *Duration error of false negative cfpsD ($ecfpsD^{FN}$):* The total duration of false negative $fpsD$ that a continuous test suggested is identical to the duration error of false negative $cfpsD$, that is, $cfpsD^{FN} = ecfpsD^{FN}$. However, the absolute total duration of a continuous test incorrectly indicating temporary control violation provides

only limited information because it lacks context. Therefore, we also calculate the ratio between $ecfpsD^{FN}$and the total amount of time during which the continuous test indicated that the cloud service does not satisfy a control ($ecfpsD^{FN} + ecfpsD^{TN}$):

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

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

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

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

We can use $ecfpsD_{rel}^{FP}$ to describe the proportion of the duration of control violation events which remained undetected in total.

7.4.4.3 Precision measures based on $cfpsD$

According to our definition of precision in Section 7.1.2, precision refers to closeness of agreement between successively measured values which implies that precision measures need at least two measured values as input. Since $ecfpsD^{TN}$, $ecfpsD^{FN}$, and $ecfpsD^{FP}$ are exactly calculated once after experimental evaluation of a continuous test, we cannot apply the concept of precision using $cfpsD$.

7.5 Example evaluation of continuous test results

In this section, we present three scenarios in which we apply our method introduced in the previous sections to evaluate and compare continuous test results. In the next section, we describe the components of our experimental setup. Thereafter, we present three scenarios in which cloud service providers seek to evaluate tests to support continuous certification of cloud services according to controls related to the properties availability and security.

7.5.1 Setup and environment

In this section, we describe the experimental setup which we use to evaluate results produced by the continuous tests. We begin with the cloud services which are subject to testing. Then we outline the control violation framework which is used to manipulate properties of the cloud services under test so that they violate one or more controls (Section 7.5.1.2) as well as the continuous tests' implementations (Section 7.5.1.3). Finally, we introduce the evaluation engine (Section 7.5.1.4) which computes the accuracy and precision measures presented in Section 7.4.

7.5.1.1 Cloud services under test

We test instances of IaaS which are provided by OpenStack Mitaka[112] ($IaaS^{OS}$). Each test presented in Section 7.5.2, 7.5.3 and 7.5.4 is executed on an individual instance of $IaaS^{OS}$. In total, three virtual machines are used each of equipped with 2 VCPUs and 4 GB RAM and running Ubuntu 16.04 server. Furthermore, all three instances are attached to the same tenant network. The two instances which are used to continuously test security communication configuration (Section 7.5.3) and secure interface configuration (Section 7.5.4) are additionally running an Apache[113] web server and a MongoDB[114] database which we refer to as $SaaS^{OS}$ and $PaaS^{OS}$, respectively.

7.5.1.2 Control violation framework

In order to trigger control violation events, we developed a lightweight framework in Java which allows to continuously manipulate properties of a cloud service under test as described in Section 7.3.1. The framework is extensible, that is, novel control violations can be added. Furthermore, multiple control violation sequences can be executed concurrently.

The framework persists each control violation event including start and end time of each event, event duration and current iteration. This data establishes the ground truth which is later used by the evaluation engine (see Section 7.5.1.4) to evaluate the accuracy and precision of continuous test results. The control violation framework is deployed on a designated virtual machine. This machine is attached to the identical tenant network as $IaaS^{OS}$, $PaaS^{OS}$, and $SaaS^{OS}$.

7.5.1.3 Continuous testing framework

Tests are implemented following the framework introduced in Chapter 4. The tests are deployed on an external host, attached to a different network than the cloud services under test.

7.5.1.4 Evaluation engine

This component is responsible for computation of accuracy and precision measures described in Section 7.4 as well as test and control violation statistics. To that end, we employ the *Apache Commons Math* library. The evaluation engine is implemented in Java and is hosted locally on a personal computer and uses the control violation sequence's data and produced test results as input.

7.5.2 Continuously testing availability

In this scenario, we consider a cloud service provider who seeks to certify controls related to the property *availability*. As already described in Section 5.2.2.1, examples of cloud-specific controls which can be related to the availability of cloud service components are *RB-02 Capacity management – monitoring* of the Cloud Computing Compliance Controls Catalogue (BSI C5) [31], *IVS-04* of the Cloud Control Matrix (CCM) [22] as well as *A.12.1.3 Capacity management* of ISO/IEC 27001:2013 [24].

[112] `https://www.openstack.org/software/mitaka/` [Accessed: 2018-12-13]
[113] `https://httpd.apache.org/` [Accessed: 2018-12-13]
[114] `https://www.mongodb.com/` [Accessed: 2018-12-13]

The provider wants to select a continuous test which overestimates the duration of detected violations of the availability of cloud service components as little as possible. As a consequence, the number of false negative test results which a test produces incorrectly indicating an availability related control violation should be kept to a minimum.

7.5.2.1 Alternative continuous tests

The cloud provider has the following three candidates of continuous tests to choose from:

- *PingTest*: Pings $IaaS^{OS}$ where each tests sends ten ECHO_REQUEST packets, thereby repeatedly measures round trip times on the Internet Layer using ICMP packets. A test passes if the returned round trip time (rrt) satisfies the following two assertions: $assert_rtt_avg < 20ms$ and $assert_rtt_sd < 10ms$.
- *TCPTest*: Sends TCP segments to determine whether $IaaS^{OS}$ is available. We use *Nping*[115] to implement this test which sends five probes per test. A single test passes if the average response time and the maximum response time of probes are not greater than 75 and 100 ms, respectively.
- *SSHTest*: Connects to $IaaS^{OS}$ via SSH and then test the session using an SSH_MSG_CHANNEL_REQUEST. We uses the *Trilead SSH2*[116] library to establish a SSH connection. A single test passes if no I/O exception is thrown during connection establishment and session testing.

The interval of *PingTest*, *TCPTest* and *SSHTest* is configured to 60 seconds, that is, the next test started 60 seconds after the previous one completed. Further, there is no additional offset between test executions and the number of successive iterations is set to infinity. Note that only test results produced during the control violation sequence are considered for the evaluation.

7.5.2.2 Control violation configuration

We trigger 1000 downtimes of $IaaS^{OS}$ for each of the three candidate tests. Each downtime event lasted at least 60 seconds plus selecting $[0, 30]$ seconds at random ($cveD$). Furthermore, the interval between two downtime events is at least 120 seconds plus selecting $[0, 60]$ seconds at random ($cveW$).

Table 7.1 provides an overview of the three different control violation sequences used to evaluate *PingTest*, *TCPTest*, and *SSHTest*. To that end, the total downtime ($ccveD$), the mean duration of each downtime ($\bar{x}_{cveD}$) and standard deviation (sd_{cveD}) for each control violation sequence V is shown.

7.5.2.3 Test statistics

The results of *PingTest*, *TCPTest*, and *SSHTest* are summarized in Table 7.2. These include the results observed for each of the universal test metrics brC, $fpsC$, $fpsD$ and $cfpsD$ which were introduced in Section 4.3.5. Furthermore, we also include the total number of executed tests ($tsrC$) as well as the mean ($\bar{x}_{tsr}$), standard deviation (sd_{tsr}), min (min_{tsr}) and max (max_{tsr}) duration of tests.

[115] `https://nmap.org/nping/` [Accessed: 2018-12-13]
[116] `https://github.com/jenkinsci/trilead-ssh2` [Accessed: 2018-12-13]

Table 7.1: Summary of control violation statistics for PingTest, TCPTest, and SSHTest

Sequence statistic (sec)	$\mathbf{V}^{\text{PingTest}}$	$\mathbf{V}^{\text{TCPTest}}$	$\mathbf{V}^{\text{SSHTest}}$
$ccveD$	75102.11	75544.07	75706.71
$\bar{x}_{cveD}$	75.11	75.54	75.71
sd_{cveD}	8.97	8.70	8.82
min_{cveD}	60.17	60.38	60.39
max_{cveD}	90.33	90.6	92.58

7.5.2.4 Accuracy and precision of PingTest, TCPTest and SSHTest

This section presents the accuracy and precision of PingTest, TCPTest as well as SSHTest observed during evaluation.

Accuracy and precision based on Basic-Result-Counter (*brC*) Table 7.3 shows the results of evaluating PingTest, TCPTest and SSHTest based on the Basic-Result-Counter (brC) test metric. Recall that in our scenario, the cloud service provider desires a continuous test which produces as little as possible false negative results. Therefore, we select those accuracy and precision measures to evaluate PingTest, TCPTest, and SSHTest which take false negative basic results (brC^{FN}) into account. These include (for definition of the measures see Section 7.4.1.2 and 7.4.1.3): Overall accuracy (oac^{brC}), false negative rate (fnr^{brC}), false omission rate (for^{brC}) and negative predictive value (npv^{brC}). For improved readability, these measures are highlight in Table 7.3.

TCPTest has the highest overall accuracy (97.52%) while SSHTest has the lowest (80.13%). Yet SSHTest has a perfect false negative rate (fnr), a perfect false omission rate (for) and a perfect negative predictive value (npv). The reason for this is that SSHTest did not produce any false negative basic test results during the evaluation (Table 7.2). Thus, if the cloud provider was only to rely on the accuracy and precision derived from the brC test metric, then the provider would select SSHTest.

Accuracy and precision based on Failed-Passed-Sequence-Counter (*fpsC*) The results of evaluating PingTest, TCPTest, and SSHTest based on the Failed-Passed-Sequence-Counter ($fpsC$) are presented in Table 7.4. As already mentioned above, in our scenario, the cloud provider is interested in the continuous test which produces the lowest number of false negative results. In context of accuracy and precision measures derived from $fpsC$, we thus select false omission rate (for^{fpsC}) and negative predictive value (npv^{fpsC}) – introduced in Section 7.4.2.2 and 7.4.2.3 – because both consider false negative fps.

PingTest has the best evaluation results since it has the highest negative predictive value (99.69%) and the lowest false omission rate (0.31%). Furthermore, SSHTest has the worst results since it has the lowest npv (97.9%) and the highest for (2.1%). The reason for this evaluation result of SSHTest can easily be observed in Table 7.2: SSHTest produced the highest number of false negative fps during evaluation. Therefore, if the cloud provider was to only rely on accuracy and precision derived from the $fpsC$ test metric, then the provider would select PingTest.

Table 7.2: Summary of test statistics of PingTest, TCPTest, and SSHTest

Test statistic		**PingTest**	**TCPTest**	**SSHTest**
	$tsrC$	3153	3546	2491
tsr (sec)	$\bar{x}_{tsr}$	12.04	4.38	30.91
	sd_{tsr}	4.51	0.47	44.90
	min_{tsr}	9.01	4.01	0.54
	max_{tsr}	20.03	5.22	127.34
brC	brC^{TP}	2024	2316	1488
	brC^{FP}	2	3	11
	brC^{TN}	795	1067	0
	brC^{FN}	2	6	0
	brC^{PTN}	207	74	508
	brC^{PFP}	122	79	484
fpsC	$fpsC^{TN}$	965	978	466
	$fpsC^{FN}$	3	9	10
	$fpsC^{FP}$	33	13	524
fpsD (sec)	$\bar{x}_{TN}$	81.22	75.13	176.74
	$sd_{fpsD^{TN}}$	13.54	23.51	61.04
	$median_{fpsD^{TN}}$	79.10	65.08	187.24
	$min_{fpsD^{TN}}$	69.02	64.06	123.09
	$max_{fpsD^{TN}}$	158.27	130.29	561.76
	$\bar{x}_{FN}$	98.79	122.17	187.29
	sd_{FN}	43.02	50.31	0.04
	$median_{fpsD^{FN}}$	79.10	129.18	187.30
	$min_{fpsD^{FN}}$	69.12	65.03	187.20
	$max_{fpsD^{FN}}$	148.13	194.35	187.34
	$\bar{x}_{FP}$	63.74	62.98	73.13
	sd_{FP}	2.63	4.13	8.35
	$median_{fpsD^{FP}}$	63.40	61.43	72.48
	$min_{fpsD^{FP}}$	60.37	60.04	60.38
	$max_{fpsD^{FP}}$	69.66	75.43	92.58
cfpsD (sec)	TN	78378.36	73481.13	82362.03
	FN	296.37	1099.50	1872.90
	FP	2103.27	818.69	38319.32

Accuracy and precision based on Failed-Passed-Sequence-Duration ($fpsD$) When evaluating PingTest, TCPTest, and SSHTest based on Failed-Passed-Sequence-Duration ($fpsD$) test metric, we obtain the results shown in Table 7.5. On average, PingTest produces true negative fps with the lowest relative error ($efpsD^{TN}_{rel}$) when estimating the duration of a downtime event (13.22%), followed by TCPTest (19.77%) and SSHTest (126.27%). Therefore, if we were to focus solely on the accuracy of the continuous test, then PingTest would be the preferable choice.

However, recall that the assumptions of our scenario state that the cloud provider seeks for a continuous test that overestimates downtime events as little as possible. To that end, we have to consider the duration error of true negative fps, that is, $efpsD^{TN}$. As pointed out in Section 7.4.3.1, $efpsD^{TN}$ is *not* based on the absolute difference between a control violation event and a corresponding $fpsD^{TN}$. Consequently, it allows us to observe whether a true negative fps overestimates or underestimates a control violation event.

Comparing $efpsD^{TN}$ of PingTest, TCPTest and SSHTest (Table 7.5) reveals that TCPTest underestimates the downtime events on average by 569 ms while PingTest and SSHTest overestimate downtime events by 5749 ms and 98050 ms, respectively. From the perspective of the cloud provider within our scenario, TCPTest thus is preferable. This choice is further supported by Figure 7.20 which allows for a more detailed analysis of $efpsD^{TN}$: It shows three box plots[117] each of which describes the variation of duration error of true negative fps. It is evident that duration error of true negative fps produced by SSHTest is not only higher on average but also has a higher variability. Furthermore, although PingTest and TCPTest have produced distributions of $efpsD^{TN}$ with similar variability, it is evident that the average and median of TCPTest indicate that TCPTest underestimates downtime events while PingTest overestimates them.

Accuracy and precision based on Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) Table 7.6 shows the results of evaluating PingTest, TCPTest and SSHTest based on the Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) test metric. In our scenario, the cloud provider wants to select the test which overestimates the total downtime ($cveD$) as little as possible. To that end, we will consider the total duration error of true negative fps, that is, $ecpfsD^{TN}$ and the total duration error of false negative fps, i.e., $ecfpsD^{FN}$.

Figure 7.21 illustrates the results of $ecpfsD^{TN}$ and $ecpfsD^{FN}$: TCPTest underestimates the total downtime by 2062.93 seconds while PingTest and SSHTest both overestimate the accumulated duration of downtime event by 3276.25 and 6655.32 seconds, respectively. Furthermore, SSHTest has the highest total duration of false negative fps (1872.89 seconds), followed by TCPTest (1099.5 seconds) and PingTest (296.37 seconds).

Conclusion In conclusion, we argue that our cloud service provider will *not* select SSHTest because in comparison with PingTest and TCPTest, its inaccuracy leads to the highest overestimates regarding both individual and total duration of downtime events. This leaves us with the question of whether to choose PingTest or TCPTest. The cloud provider will select TCPTest since it on average ($efpsD^{TN}$) and in total ($ecpfsD^{TN}$) underestimates the duration of downtime events. Although PingTest produces the lowest total false negative duration ($ecpfsD^{FN}$), it cannot compensate for overestimating the total duration of true

[117]Each box shows the interquartile range (IQR) of the distribution and the whiskers display lowest and highest data points within $1.5 \times IQR$ in the lower and upper quartile, respectively. Furthermore, outliers are indicated by small circles while the average and median are displayed by a dashed green line and a solid red line, respectively.

Table 7.3: Evaluation of PingTest, TCPTest and SSHTest to test availability of $IaaS^{OS}$ based on the Basic-Result-Counter (brC) test metric

Accuracy & precision measure		PingTest	TCPTest	SSHTest
ebrC (%)	oac	96.0	97.52	80.13
	$E^{95\%}_{oac}$	0.68	0.51	1.57
	tnr	88.99	93.30	50.65
	$E^{95\%}_{tnr}$	1.83	1.40	3.09
	tpr	99.90	99.74	100
	$E^{95\%}_{tpr}$	0.14	0.21	0
	fnr	0.20	0.53	0
	$E^{95\%}_{fnr}$	0.26	0.41	0
	fpr	11.01	6.70	49.35
	$E^{95\%}_{fpr}$	1.82	1.40	3.09
	fdr	5.77	3.42	24.96
	$E^{95\%}_{fdr}$	0.99	0.73	1.90
	ppv	94.23	96.58	75.04
	$E^{95\%}_{ppv}$	0.99	0.73	1.90
	for	0.20	0.52	0
	$E^{95\%}_{for}$	0.28	0.42	0
	npv	99.8	99.48	100
	$E^{95\%}_{npv}$	0.28	0.42	0

negatives: If we add $ecpfsD^{FN}$ to $ecpfsD^{TN}$ which gives us the total duration of all observed negative fps, for PingTest we obtain

$$ecpfsD^{TN} + ecpfsD^{FN} = 3276.25sec + 296.37sec = 3572.62sec$$

and for TCPTest we compute

$$ecpfsD^{TN} + ecpfsD^{FN} = -2062.93sec + 1099.5sec = -963.43sec.$$

It is evident that TCPTest still dominates PingTest in context of our scenario since it underestimates downtime events rather than overestimating them. Consequently, the cloud provider will select TCPTest to determine the availability of her cloud service components.

7.5.2.5 Calibrating precision

Table 7.3 includes margins of error, e.g., $E^{95\%}_{oac}$, $E^{95\%}_{tnr}$, $E^{95\%}_{tpr}$ and $E^{95\%}_{ppv}$. We use these margins of error to construct interval estimates describing the precision of a continuous test based on the brC test metric. As described in Section 7.1.2.5, we can treat the sample size n as a parameter to calibrate our evaluation according to the desired precision, that is, a desired margin of error $\hat{E}$. Within the following two paragraphs, we exemplify how to calibrate the experimental evaluation according to a desired precision for *PingTest* and *TCPTest*.

Table 7.4: Evaluation of PingTest, TCPTest and SSHTest to test availability of $IaaS^{OS}$ based on the Failed-Passed-Sequence-Counter ($fpsC$) test metric

Accuracy & precision measure		**PingTest**	**TCPTest**	**SSHTest**
efpsC (%)	tnr	96.69	98.69	47.07
	$E^{95\%}_{tnr}$	1.11	0.71	3.11
	fpr	3.31	1.31	52.93
	$E^{95\%}_{fpr}$	1.11	0.71	3.11
	for	0.31	0.91	2.10
	$E^{95\%}_{for}$	0.35	0.59	1.29
	npv	99.69	99.09	97.90
	$E^{95\%}_{npv}$	0.35	0.59	1.29

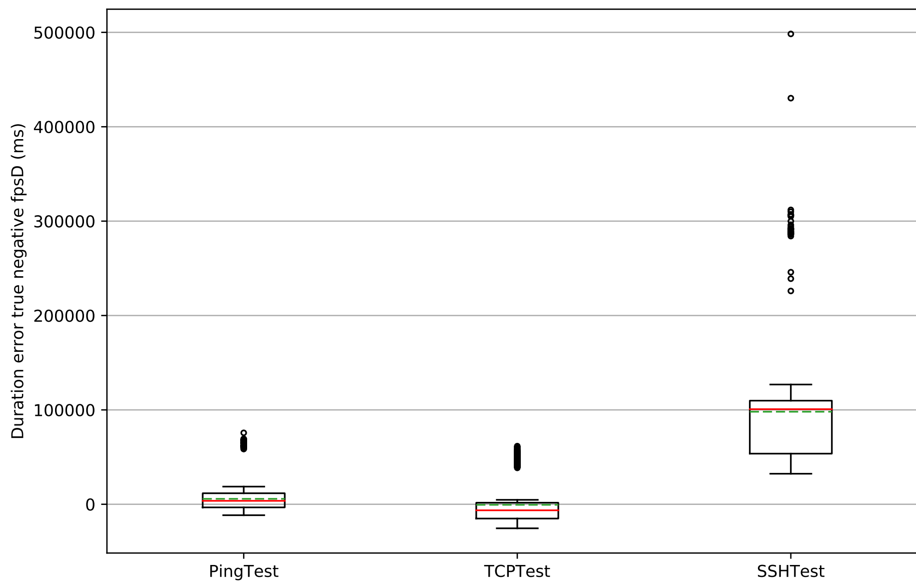


Figure 7.20: Duration error of true negative fps ($efpsD^{TN}$) of PingTest, TCPTest, and SSHTest

Table 7.5: Evaluation of PingTest, TCPTest and SSHTest to test availability of $IaaS^{OS}$ based on the Failed-Passed-Sequence-Duration ($fpsD$) test metric

Accuracy & Precision measure		PingTest	TCPTest	SSHTest
$efpsD^{TN}$ (ms)	$\bar{x}$	5749	-569	98050
	$\widehat{x}$	3643	-6386.5	100754
	sd	14143	21996	61586
	min	-11586	-25467	32372
	max	75692	61613	498360
	$E^{95\%}$	895	1378	5606
$efpsD^{TN}_{rel}$ (%)	$\bar{x}$	13.22	19.77	126.27
	$\widehat{x}$	9.22	14.94	116.49
	sd	14.74	18.41	83.69
	min	0.26	0.32	35.66
	max	119.37	91.26	786.01
	$E^{95\%}$	0.93	1.16	7.62
$efpsD^{TN}_{pre}$ (ms)	$\bar{x}$	33102	32364	6026
	$\widehat{x}$	32087	32103.5	15035.5
	sd	19634	18293	54703
	min	-8342	132	-368338
	max	69584	65098	59740
	$E^{95\%}$	1240	1148	4980
$efpsD^{TN}_{post}$ (ms)	$\bar{x}$	38852	31795	104077
	$\widehat{x}$	38223	31250.5	110652
	sd	19777	18584	23052
	min	153	48	60442
	max	70141	64082	161631
	$E^{95\%}$	1249	1166	2098
$efpsD^{FN}$ (ms)	$\bar{x}$	98789	122166	187289
	$\widehat{x}$	79110	129178	187296
	sd	43023	50306	42
	min	69124	65030	187196
	max	148132	194353	187336
	$E^{95\%}$	106876	38668	30
$efpsD^{FP}$ (ms)	$\bar{x}$	63736	62976	73128
	$\widehat{x}$	63400	61430	72481.5
	sd	2626	4131	8345
	min	60368	60404	60380
	max	69657	75430	92584
	$E^{95\%}$	931	2496	716

Table 7.6: Evaluation of PingTest, TCPTest and SSHTest to test availability of $IaaS^{OS}$ based on the Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) test metric

$ecfpsD^{TN}$	TN(ms)	3276253	-2062934	6655321
	TN(%)	4.36	2.73	8.79
$ecfpsD^{FN}$	FN(ms)	296366	1099497	1872887
	FN(%)	0.38	1.47	2.22
$ecfpsD^{FP}$	FP(ms)	2103272	818685	38319324
	FP(%)	2.80	1.08	50.62

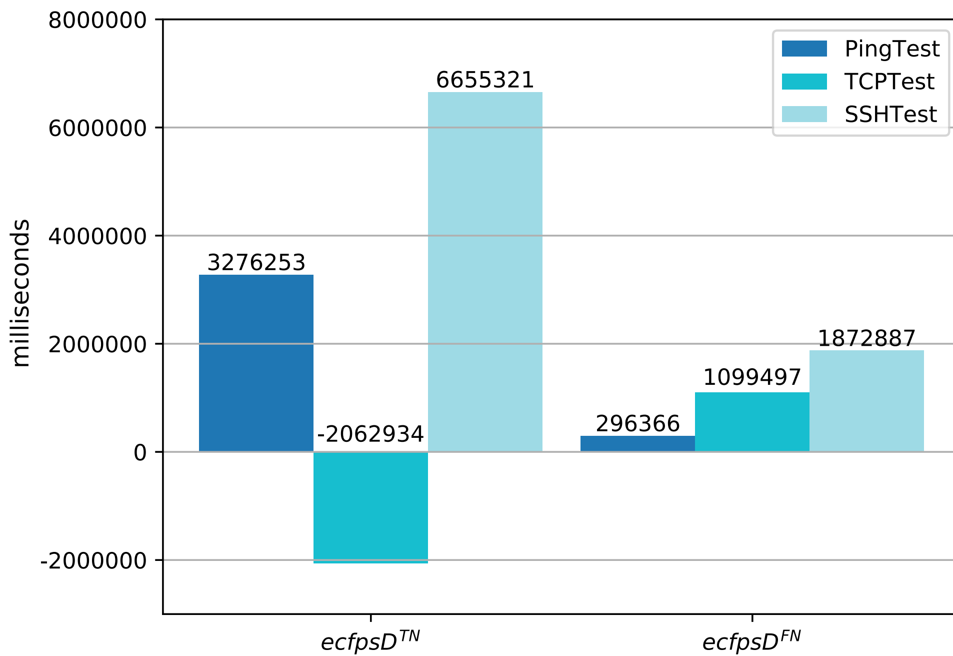


Figure 7.21: Total duration error of true negative fps ($ecfpsd^{TN}$) and false negative fps ($ecfpsd^{FN}$) of PingTest, TCPTest, and SSHTest

Considering TCPTest, the sum of observed brC^{TN}, brC^{PTN}, brC^{FP} and brC^{PFP} equals 1223 brC (Table 7.2). On this basis, we compute the 95% confidence level for the true negative rate tnr, that is, 93.3% ± 1.4. Assuming that this observed value of E_{tnr} is our desired one, then we compute the required sample size n for confidence intervals for proportions using formula 7.1, that is,

$$\tilde{n} = \frac{z_{95\%} \times \widehat{tnr} \times (1 - \widehat{tnr})}{E^2} = \frac{1.96 \times 0.933 \times (1 - 0.933)}{0.014^2} \approx 626.$$

This means that we have to trigger as many control violation events – here downtime events – to observe $brC^{TN} + brCP^{TN} + brC^{FP} + brC^{PFP} = 626\, brC$. Since we observed the sum of 1223 brC, we can state that we are 95% confident that *TCPTest* – with respect to the selected test and control violation configurations – has a true negative rate between 91.9 and 94.7%. Calibrating the precision of evaluations of continuous tests based on the $fpsC$ test metric works analogously.

Let's now consider *PingTest* to calibrate the precision of a continuous test based on the $fpsD$ test metric: We observed 965 true negative fps (Table 7.2) which are used to compute a 95% confidence level for the mean ($\bar{x}$) of the duration error of true negative fps ($efpsd^{TN}$). The margin of error ($E^{95\%}$) is 895 ms leading to following confidence interval (Table 7.5): 5749 ms±895.

In Section 7.1.2.5 we pointed out that we can determine the required minimum sample size for a desired margin of error using formula 7.4. Let us first consider that the observed value for the margin of error coincides with our desired value for $E^{95\%}$. In this case, we simply plug in the observed value for $E^{95\%}_{TN}$ and the standard deviation from *PingTest* as an estimate for the population variance (σ^2). As a result, we obtain

$$\tilde{n} = \frac{\sigma^2 \times z^2}{E^2} = \frac{14143^2 \times 1.96^2}{895^2} \approx 960$$

required true negative fps. This means that we have to trigger as many downtime events as are necessary to observe at least 960 fps^{TN}. As an alternative case, consider that our desired margin of error is 1000 ms. In order to achieve this precision, ≈ 769 fps^{TN} are required to be observed during evaluation. Having observed 965 true negative fps, we can state that we are 95% confident that the duration error ($efpsd^{TN}$) which PingTest makes on average when estimating the duration of a downtime event – with respect to the chosen test and control violation configuration – is between 4854 and 6644 ms.

7.5.3 Continuously testing secure communication configuration

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

The SaaS provider seeks certification of his SaaS application according to controls which are related to property *secure communication configuration*. As already discussed in Section 5.4.2.1, examples for cloud-specific controls which are related to the secure communication configuration of cloud services are *KRY-02 Encryption of data for transmission (transport*

encryption) of the Cloud Computing Compliance Controls Catalogue (BSI C5) [31], *EKM-03: Encryption & Key Management Sensitive Data Protection* of CSA's Cloud Control Matrix (CCM) [22], and *A.14.1.2 Securing application services on public networks* of ISO/IEC 27001:2013 [24].

To support certification of his SaaS application, the SaaS provider want to utilize a continuous test and configure it in such a manner that it most accurately indicates if the secure communication configuration property of his SaaS application does not hold. This implies that the test should ideally detect any violation of the secure communication configuration property while the number of false positive results produced by the test should be as low as possible. Furthermore, if an insecure communication configuration has been detected, then the SaaS provider seeks a test configuration which as accurately as possible detects how long it takes the PaaS provider to fix vulnerable communication configurations.

7.5.3.1 Alternative test configurations

We make use of the tool *sslyze*[118] to analyze the TLS configuration of $SaaS^{OS}$. The output of sslyze is inspected to discover whether the web server offers to communicate via known vulnerable cipher suites. In case the web server does offer support for vulnerable cipher suites, the secure communication configuration property of $SaaS^{OS}$ does not hold, leading to a violation of certificates' controls relating to this property.

The SaaS provider within our scenario can choose between the following three different candidate configurations for the test *TLSTest*:

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

Furthermore, no additional offset between test executions is configured and the number of successive iterations for all three TLSTest variants is set to infinity. Note that only test results produced during the control violation sequence are considered for evaluation.

7.5.3.2 Control violation configuration

In order to evaluate the three candidate configurations of TLSTest, we triggered 1000 vulnerable TLS configurations of $SaaS^{OS}$ for each TLSTest variants. These vulnerable TLS configurations consist of manipulating the web server configuration of $SaaS^{OS}$ such that it supports TLS communication using the weak cipher suite TLS_RSA_WITH_DES_CBC_SHA. Each event where the TLS configuration is not secure lasted at least 60 seconds plus selecting $[0, 30]$ seconds at random. The interval between consecutive vulnerable configuration events was at least 120 seconds plus selecting $[0, 60]$ seconds at random. Table 7.7 provides an overview of the control violation sequence statistics observed during experimental evaluation of $TLSTest^{[0,10]}$, $TLSTest^{[0,30]}$ and $TLSTest^{[0,60]}$.

[118] `https://github.com/nabla-c0d3/sslyze` [Accessed: 2018-12-13]

Table 7.7: Summary of control violation sequence statistics for TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$, and TLSTest$^{[0,60]}$

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

7.5.3.3 Test statistics

The results produced by TLSTest are shown in Table 7.8 consisting of the values observed for each of the universal test metrics introduced in Section 4.3.5. Moreover, the total number of executed tests ($tsrC$) as well as the mean ($\bar{x}_{tsr}$), standard deviation (sd_{tsr}), min (min_{tsr}) and max (max_{tsr}) duration of tests are included. Note that for each TLSTest variant, we only observed a single value for false positive $fpsD$ (i.e., $fpsC^{FP} = 1$) and thus we cannot compute average ($\bar{x}_{FP}$), median ($median_{fpsD^{FP}}$), and standard deviation (sd_{FP}) for TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$, and TLSTest$^{[0,60]}$. The corresponding fields of Table 7.8 are marked as *na* which is short for *not applicable*.

7.5.3.4 Accuracy and precision of TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$ and TLSTest$^{[0,60]}$

This section presents the accuracy and precision of TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$ and TLSTest$^{[0,60]}$ observed during evaluation.

Accuracy and precision based on Basic-Result-Counter (*brC*) Table 7.9 shows the results of evaluating TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$ and TLSTest$^{[0,60]}$ based on the Basic-Result-Counter (*brC*) test metric. Recall that according to the constraints of our scenario, the SaaS provider wants to select a configuration of TLSTest which produces the minimum number of false positive basic test results (brC^{FP}). Through inspecting Table 7.8, we can easily observe that TLSTest$^{[0,10]}$ produced the highest number of brC^{FP}, followed by TLSTest$^{[0,30]}$ and TLSTest$^{[0,60]}$. However, only relying on these absolute counts of brC^{FP} is misleading since TLSTest$^{[0,10]}$ executed more than twice as many tests as TLSTest$^{[0,30]}$ which is not reflected by the mere absolute number of brC^{FP}. Consequently, we have to make use of the accuracy and precision measures introduced in Section 7.4.1.2 and 7.4.1.3 which relate brC^{FP} to the remaining results of the continuous test evaluation. These are: Overall accuracy (oac^{brC}), true negative rate (tnr^{brC}), false positive rate (fpr^{brC}), false discovery rate (fdr^{brC}) and positive predictive value (ppv^{brC}). These measures are highlighted in Table 7.9 for improved readability.

TLSTest$^{[0,10]}$ has the lowest overall accuracy (98.24%) and the lowest true negative rate (97.06%). Further, TLSTest$^{[0,10]}$ has the highest false discovery rate (1.55%), followed by TLSTest$^{[0,60]}$ (1.46%) and TLSTest$^{[0,30]}$ (1.34%). However, we argue that the most suitable accuracy measure in context of our scenario is the false positive rate since it captures the ratio between incorrectly passed tests and all test that were expected to fail: TLSTest$^{[0,10]}$ has the highest fpr (2.94%), followed by TLSTest$^{[0,60]}$ (2.84%) and TLSTest$^{[0,30]}$ (2.64%).

Table 7.8: Summary of test statistics of TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$, and TLSTest$^{[0,60]}$

Test statistic		**TLSTest**		
		[0,10]	**[0,30]**	**[0,60]**
	$tsrC$	34801	13771	7332
tsr (sec)	$\bar{x}_{tsr}$	1.50	1.40	1.38
	sd_{tsr}	0.59	0.62	0.46
	min_{tsr}	0.10	0.10	0.10
	max_{tsr}	19.73	19.39	19.18
brC	brC^{TP}	22484	9024	4793
	brC^{FP}	8	5	3
	brC^{TN}	11585	4504	2410
	brC^{FN}	260	83	39
	brC^{PTN}	106	33	18
	brC^{PFP}	346	118	68
fpsC	$fpsC^{TN}$	871	893	969
	$fpsC^{FN}$	184	110	30
	$fpsC^{FP}$	1	1	1
fpsD (sec)	$\bar{x}_{TN}$	74.78	75.41	75.59
	$sd_{fpsD^{TN}}$	9.94	13.93	22.96
	$median_{fpsD^{TN}}$	74.55	75.43	75.30
	$min_{fpsD^{TN}}$	41.75	39.64	18.20
	$max_{fpsD^{TN}}$	97.99	114.76	138.10
	$\bar{x}_{FN}$	52.32	73.46	69.66
	sd_{FN}	32.52	19.29	29.55
	$median_{fpsD^{FN}}$	66.17	73.54	65.87
	$min_{fpsD^{FN}}$	0.40	13.09	24.27
	$max_{fpsD^{FN}}$	96.23	114.18	143.17
	$\bar{x}_{FP}$	*na*	*na*	*na*
	sd_{FP}	*na*	*na*	*na*
	$median_{fpsD^{FP}}$	*na*	*na*	*na*
	$min_{fpsD^{FP}}$	87.02	73.02	84.02
	$max_{fpsD^{FP}}$	87.02	73.02	84.02
cfpsD (sec)	TN	65136.55	67340.92	73245.40
	FN	9627.13	8080.22	2089.73
	FP	87.02	73.02	84.02

As a consequence, the SaaS provider will favor TLSTest$^{[0,30]}$ if he was only to draw on the accuracy derived from the brC test metric.

Accuracy and precision based on Failed-Passed-Sequence-Counter ($fpsC$) The results of evaluating TLSTest$^{[0,10]}$,TLSTest$^{[0,30]}$, and TLSTest$^{[0,60]}$ based on the Failed-Passed-Sequence-Counter ($fpsC$) are presented in Table 7.10. As already pointed out in the previous paragraph, the SaaS provider within our scenario seeks to configure TLSTest such that it produces the lowest number of false positive results possible. In context of accuracy and precision measures which are based on the $fpsC$ test metric, we therefore select the false positive rate (fpr^{fpsC}) and the true negative rate (tnr^{fpsC}) – defined in Section 7.4.2.2 and 7.4.2.3 – to evaluate the variants of TLSTest since they tell us – out of all events that should have been detected – how many control violation events were correctly detected (tnr^{fpsC}) and how many control violation events were not detected (fpr^{fpsC}).

Although each of the TLSTest variants only produces a single false positive fps (Table 7.8), TLSTest$^{[0,60]}$ has the lowest fpr (0.1%) and the highest tnr (99.9%). The reason for this is that TLSTest$^{[0,60]}$ produced a higher number of true negative fps (969) than both TLSTest$^{[0,10]}$ (871) and TLSTest$^{[0,30]}$ (893). Hence, if the SaaS provider in our scenario only relies on the accuracy derived from the $fpsC$ test metric, then he will choose TLSTest$^{[0,60]}$.

Accuracy and precision based on Failed-Passed-Sequence-Duration ($fpsD$) Evaluating TLSTest$^{[0,10]}$,TLSTest$^{[0,30]}$, and TLSTest$^{[0,60]}$ based on the Failed-Passed-Sequence-Duration ($fpsD$) produces the results shown in Table 7.11. Note that since we only observed a single false positive $fpsD$ (i.e., $fpsC^{FP} = 1$, see Table 7.8) for each TLSTest variant, we cannot compute average ($\bar{x}$), median ($\hat{x}$), standard deviation (sd) and margin of error ($E^{95\%}$) of $efpsD^{FP}$ for TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$, and TLSTest$^{[0,60]}$. The corresponding fields of Table 7.11 are marked as *not applicable (na)*.

Aside from choosing a configuration for TLSTest which produces the lowest false positive results, the SaaS provider in our scenario prefers the TLSTest variant which as accurately as possible estimates how long it took the PaaS provider to fix a detected, vulnerable communication configuration. Put differently: The SaaS provider favors a configuration of TLSTest which most accurately estimates the duration of a control violation event.

Figure 7.22 shows three box plots which describe the variation of relative duration error of true negative fps ($efpsD_{rel}^{TN}$): It is apparent that the relative error each test of TLSTest$^{[0,60]}$ makes when estimating the duration of a vulnerable communication configuration event has the greatest mean (dashed green line, 22.96%), greatest median (solid red line, 20.56%) as well as the greatest variability. Moreover, on average, TLSTest$^{[0,10]}$ produces true negative fps with the lowest relative error ($efpsd_{rel}^{TN}$) when estimating the duration of a vulnerable communication configuration event (4.56%), followed by TLSTest$^{[0,30]}$ (11.33%). Hence, in context of our scenario, the SaaS provider will select TLSTest$^{[0,10]}$ because it provides the most accurate estimate of how long it takes the PaaS provider to fix a vulnerable TLS setup.

Accuracy and precision based on Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) Table 7.12 shows the results of evaluating TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$ and TLSTest$^{[0,60]}$ based on the Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) test metric. The results of the total duration error of true negative fps ($ecfpsD^{TN}$) indicate that each variant of TLSTest underestimates the accumulated duration of vulnerable TLS configuration events. Based on $ecfpsD^{TN}$, the most accurate result is produced by TLSTest$^{[0,60]}$ (-2232.09 seconds),

followed by TLSTest$^{[0,30]}$ (-7476.24 seconds) and TLSTest$^{[0,10]}$ (-9974.22 seconds). However, in context of our scenario, the accumulated duration of true negative fps is *not* to be considered since the SaaS provider's focus lies on correctly detecting temporary vulnerable TLS configurations and estimating their individual duration. Hence, the accumulated duration of fps and thus accumulated error of fps does not affect the decision of the SaaS provider.

Conclusion We argue that the SaaS provider in our example scenario will select TLSTest$^{[0,60]}$ because the accuracy and precision measures $efpsC$ indicate that it has the highest number of correctly detected control violations, that is, TLSTest$^{[0,60]}$ produced the highest number of true negative fps. One counterargument to this conclusion is that TLSTest$^{[0,10]}$ is more accurate in estimating the duration of a vulnerable TLS configuration event (see accuracy and precision measures $efpsD$). However, we assume that it is more important to the SaaS provider in our scenario that the continuous test detects the number of occurrences of control violations most accurately than it is to most accurately estimate the duration of correctly detected violations. Note that – opposed to our scenario constraints – if we were to also compare the accuracy based on the cumulative error of true negative fps ($ecfpsD^{TN}$), then this would further foster our conclusion because TLSTest$^{[0,60]}$ produces the lowest value for $ecfpsD^{TN}$.

Table 7.9: Evaluation of TLSTest to test secure communication configuration of $SaaS^{OS}$ based on the Basic-Result-Counter (brC) test metric

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

Table 7.10: Evaluation of TLSTest to test secure communication configuration of *SaaS*^{OS} based on the Failed-Passed-Sequence-Counter ($fpsC$) test metric

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

7.5.4 Continuously testing secure interface configuration

Similar to the previous scenario, in this scenario we consider a SaaS provider who uses database services provided by a PaaS provider. The SaaS provider has no access to the components used to provide the database other than the APIs that allows to persist and retrieve data as well as managing access to the database. Different to the scenario described in the previous section, the SaaS provider in this scenario is responsible for configuring who can access the database.

The SaaS provider wants to certify his SaaS application according to controls related to the property *secure interface configuration*. As already described in Section 5.5.2.1, examples of cloud-specific controls related to secure interface configuration are *IVS-06: Infrastructure & Virtualization Security Network Security* and *IVS-07: Infrastructure & Virtualization Security OS Hardening and Base Controls* of CSA's Cloud Control Matrix (CCM) [22] as well as *RB-22 Handling of vulnerabilities, malfunctions and errors – system hardening* of the BSI C5 [31].

Similar to the previous scenario, the SaaS provider seeks to configure a continuous test which most accurately indicates if and how many temporary, vulnerable interface configuration event have occurred. However, different to the previous scenario, this SaaS provider is looking to find a test configuration which has the highest number of true results, i.e., true positive or true negative results. Furthermore, since in this scenario it is upon the SaaS provider to fix a vulnerable interface configuration of the used database service, the SaaS provider neglects how long it takes to fix a vulnerable interface configuration. This means that accuracy and precision of estimating individual and cumulative duration of vulnerable interface configuration events do not impact on the decision of the provider as to which test configuration to select.

7.5.4.1 Alternative continuous test configurations

In order to detect vulnerable interfaces, we make use of the tool *Nmap*[119] to discover reachable ports of *PaaS*^{OS} in the range 1-65535 (*PortTest*). The output of Nmap is parsed and compared

[119] `https://nmap.org/` [Accessed: 2018-12-13]

Table 7.11: Evaluation of TLSTest to test secure communication configuration of $SaaS^{OS}$ based on the Failed-Passed-Sequence-Duration ($fpsD$) test metric

Test statistic		**TLSTest**		
		[0,10]	**[0,30]**	**[0,60]**
$\boldsymbol{efpsD^{TN}}$ **(ms)**	$\bar{x}$	-52	644	151
	$\widehat{x}$	254	603	-442
	sd	4508	10465	20991
	min	-22201	-40073	-51054
	max	11552	25906	51510
	$E^{95\%}$	300	687	1323
$\boldsymbol{efpsD^{TN}_{rel}}$ (%)	$\bar{x}$	4.56	11.33	22.96
	$\widehat{x}$	3.54	9.62	20.56
	sd	4.15	8.68	16.45
	min	0.01	0.01	0.01
	max	31.58	48.86	80.83
	$E^{95\%}$	0.28	0.57	1.04
$\boldsymbol{efpsD^{TN}_{pre}}$ **(ms)**	$\bar{x}$	4677	10587	21490
	$\widehat{x}$	4030	9308	19383
	sd	3759	7682	14688
	min	-1582	-774	44
	max	25739	45251	61204
	$E^{95\%}$	250	505	925
$\boldsymbol{efpsD^{TN}_{post}}$ **(ms)**	$\bar{x}$	4624	11230	21641
	$\widehat{x}$	4217	10154	19407
	sd	2502	6999	14098
	min	18	31	18
	max	15350	29288	58807
	$E^{95\%}$	166	460	889
$\boldsymbol{efpsD^{FN}}$ **(ms)**	$\bar{x}$	52321	73457	69658
	$\widehat{x}$	66173	73536.5	65874.5
	sd	32517	19295	29548
	min	396	13087	24270
	max	96231	114183	143168
	$E^{95\%}$	4730	3646	11033
$\boldsymbol{efpsD^{FP}}$ **(ms)**	$\bar{x}$	*na*	*na*	*na*
	$\widehat{x}$	*na*	*na*	*na*
	sd	*na*	*na*	*na*
	min	87024	73022	84022
	max	87024	73022	84022
	$E^{95\%}$	*na*	*na*	*na*

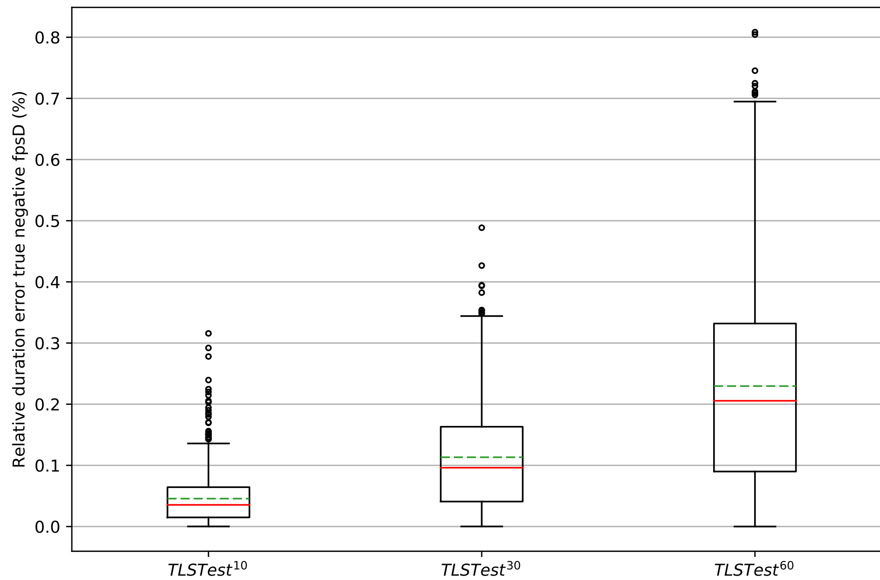


Figure 7.22: Relative duration error of true negative fps ($efpsD^{TN}_{rel}$) of TLSTest$^{[0,10]}$, TLSTest$^{[0,30]}$ and TLSTest$^{[0,60]}$

Table 7.12: Evaluation of TLSTest to test secure communication configuration of $SaaS^{OS}$ based on the Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) test metric

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

against a list of ports which should be publicly reachable. In case one or more ports are reachable which are not whitelisted, then the property secure interface configuration is not satisfied. This, in turn, leads to a violation of a certificate's controls relating to this property.

In context with our scenario, the SaaS provider can choose one of the following candidate configurations for the continuous test *PortTest*:

- *PortTest^{10}:* Each execution of PortTest is triggered statically 10 seconds after the last test execution completed.

- *PortTest^{30}:* Each execution of PortTest is triggered statically 30 seconds after the last

test execution completed.

- *PortTest^{60}:* Each execution of PortTest is triggered statically 60 seconds after the last test execution completed.

Analogous to the previous two example evaluations presented in Section 7.5.2 and 7.5.3, no additional offset between repeated test executions is configured while the number of successive iterations all three PortTest variants is set to infinity. Also, only test results produced during the start of the first and end of the last vulnerable interface configuration event (see following section for detail) are considered for evaluation.

7.5.4.2 Control violation configuration

In order to evaluate the three candidate configurations of PortTest, we triggered 1000 vulnerable interface configuration events by temporarily opening port 27018 on *PaaS^{OS}*. 27018 is the default port for sharded instances of MongoDB that should not be publicly reachable. Each vulnerable interface configuration event is set to last at least 60 seconds plus selecting [0, 30] seconds at random. Further, the interval between consecutive vulnerable interface configuration events was at least 120 seconds plus selecting [0, 60] seconds at random. Table 7.13 summarizes the control violation sequence statistics observed for each PortTest variant during experimental evaluation.

Table 7.13: Summary of control violation sequence statistics for PortTest^{10}, PortTest^{30}, and PortTest^{60}

Sequence statistic (sec)	**V$^{\text{PortTest}}$**		
	10	**30**	**60**
$ccveD$	76020.15	75418.79	75070352
$\bar{x}_{cveD}$	76.02	75.42	75.07
sd_{cveD}	8.78	9.30	8.82
min_{cveD}	60.17	60.17	60.24
max_{cveD}	90.33	90.66	90.73

7.5.4.3 Test statistics

Table 7.14 shows the results produced by PortTest^{10}, PortTest^{30} and PortTest^{60}. These include the values observed for the universal test metrics introduced in Section 4.3.5 during evaluation of the three PortTest variants. Table 7.14 also contains total number of executed tests ($tsrC$) as well as the mean (x_{tsr}), standard deviation (sd_{tsr}), min (min_{tsr}) and max (max_{tsr}) duration of tests. Note that for PortTest^{10} and PortTest^{30}, we only observed a single value for false positive $fpsD$ (i.e., $fpsC^{FP} = 1$) and thus we cannot compute average ($\bar{x}_{FP}$), median ($median_{fpsD^{FP}}$), and standard deviation (sd_{FP}) for these PortTest variants. The corresponding fields of Table 7.14 are marked as *not applicable (na)*.

7.5.4.4 Accuracy and precision of PortTest

This section presents the accuracy and precision of PortTest^{10}, PortTest^{30}, and PortTest^{60} observed during evaluation.

Table 7.14: Summary of test statistics of PortTest[10], PortTest[30], and PortTest[60]

Test statistic		**PortTest**		
		10	**30**	**60**
	$tsrC$	22039	7501	3767
tsr (sec)	$\bar{x}_{tsr}$	0.20	0.19	0.16
	sd_{tsr}	0.17	0.13	0.17
	min_{tsr}	0.10	0.10	0.10
	max_{tsr}	2.48	2.16	2.86
brC	brC^{TP}	14630	4995	2515
	brC^{FP}	86	20	12
	brC^{TN}	7265	2475	1229
	brC^{FN}	11	5	3
	brC^{PTN}	26	0	2
	brC^{PFP}	15	4	5
fpsC	$fpsC^{TN}$	988	994	995
	$fpsC^{FN}$	15	5	3
	$fpsC^{FP}$	1	1	2
fpsD (sec)	$\bar{x}_{TN}$	75.43	75.09	74.28
	$sd_{fpsD^{TN}}$	10.43	15.38	25.49
	$median_{fpsD^{TN}}$	72.99	60.97	60.17
	$min_{fpsD^{TN}}$	10.22	30.12	60.11
	$max_{fpsD^{TN}}$	96.39	92.24	121.79
	$\bar{x}_{FN}$	67.03	90.82	120.39
	sd_{FN}	24.37	0.22	0.24
	$median_{fpsD^{FN}}$	72.67	90.91	120.27
	$min_{fpsD^{FN}}$	10.25	90.43	120.22
	$max_{fpsD^{FN}}$	94.48	90.94	120.66
	$\bar{x}_{FP}$	*na*	*na*	66.25
	sd_{FP}	*na*	*na*	5.66
	$median_{fpsD^{FP}}$	*na*	*na*	66.25
	$min_{fpsD^{FP}}$	61.17	76.26	62.25
	$max_{fpsD^{FP}}$	61.17	76.26	70.25
cfpsD (sec)	TN	74526.25	74637.64	73907.46
	FN	1005.4	454.09	361.16
	FP	61167	76.26	132.5

Accuracy and precision based on Basic-Result-Counter (*brC*) Table 7.15 shows the results of evaluating PortTest^{10}, PortTest^{30}, and PortTest^{60} based on the Basic-Result-Counter (brC) test metric. Recall that in this scenario, the SaaS provider seeks to select a configuration of PortTest which has the highest number of true results. Thus we have to select those accuracy and precision measures – introduced in Section 7.4.1.2 and 7.4.1.3 – which either take true positive or true negative basic test results or both into account. To that end, we select overall accuracy (oac^{brC}), true negative rate (tnr^{brC}) and true positive rate (tpr^{brC}).

As shown in Figure 7.23, when considering the true negative rate, then PortTest^{30} produces the best result (99.04%), followed by PortTest^{60} (98.64%) and PortTest^{10} (98.63%). However, PortTest^{10} has a higher true positive rate than PortTest^{30}. Thus only taking tpr^{brC} and tnr^{brC} into account without preference for either one leaves the SaaS provider in our scenario undecided and we have to consider further accuracy and precision measures to allow for a substantiated decision. To that end, we also consider the overall accuracy because it relates both correctly passed and failed tests to all observed test results. Here, *PortTest*^{30} achieves the highest rate (99.61%) and PortTest^{10} (99.49%) is only second. Consequently, we argue that SaaS provider – if only relying on accuracy and precision derived from brC – favors PortTest^{30} over PortTest^{10}.

Accuracy and precision based on Failed-Passed-Sequence-Counter (*fpsC*) The results of evaluating PortTest^{10}, PortTest^{30} and PortTest^{60} based on the Failed-Passed-Sequence-Counter ($fpsC$) are presented in Table 7.16. In light of our scenario's assumption that the SaaS provider seeks the variant of PortTest which produces highest number of true results, we use the true negative rate (tnr^{fpsC}) and the negative predictive value (npv^{fpsC}).

Figure 7.24 shows that PortTest^{60} has the lowest true negative rate (99.8%) while both PortTest^{10} and PortTest^{30} have a better result (99.9%). Yet PortTest^{60} produces the best negative predictive value (99.7%) and PortTest^{10} has the worst result (98.5%). Similar to evaluating the PortTest variants based on brC, also in this case their is no consensus among the chosen accuracy and precision measures as to which PortTest configuration is most preferable with regards to the SaaS provider in our example scenario.

In order to be able to make a decision, we have to add another assumption about our SaaS provider: Lets assume that this provider is rather willing to tolerate errors of incorrectly indicating a vulnerable interface configuration (false negatives) then incorrect test results suggesting the interface configuration is secure. When inspecting the definition of tnr^{fpsC} and ppv^{fpsC} (see Section 7.4.2.2), then it can easily been observed that an increase in false positive $fpsC$ will lead to an decrease in the true negative rate while the negative predictive value remains unaffected. This implies that the true negative rate will drive the provider's decision which PortTest variant to use. As a consequence, the SaaS provider will either choose PortTest^{10} or PortTest^{30} since both have the same and highest tnr^{fpsC}.

Accuracy and precision based on Failed-Passed-Sequence-Duration (*fpsD*) Table 7.17 shows the evaluation results for PortTest^{10}, PortTest^{30} and PortTest^{60} based on the Failed-Passed-Sequence-Duration ($fpsD$). Note that since we only observed a single value for false positive $fpsD$ (i.e., $fpsC^{FP} = 1$, see Table 7.14) for PortTest^{10} and PortTest^{30}, we cannot compute their average ($\bar{x}$), median ($\hat{x}$), standard deviation (sd) and margin of error ($E^{95\%}$). The corresponding fields of Table 7.17 are marked as *not applicable (na)*.

As pointed out in the scenario description, we assume that our example SaaS provider does not consider the accuracy and precision of the PortTest variants when estimating the

duration of a vulnerable interface configuration event. Therefore, the discussion of evaluation results based on $fpsD$ is omitted here.

Accuracy and precision based on Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) Evaluation results for PortTest^{10}, PortTest^{30} and PortTest^{60} based on $cfpsD$ are shown in Table 7.18. However, similar to evaluating PortTest variants based on $fpsD$, the example SaaS provider does not factor in the cumulative error when estimating the cumulative duration of vulnerable interface configuration events triggered during experimental evaluation of the PortTest variants.

Conclusion We argue that the SaaS provider within our scenario will select PortTest^{30} since the accuracy measures and precision measures $ebrC$ and $efpsC$ indicate that this configuration of PortTest best satisfies our scenario's requirements. However, note that in order to arrive at this conclusion, we needed to add further assumptions to our scenario as otherwise choosing which PortTest configuration is best suited is ambiguous, i.e., there exist more than one correct solution. This reveals an important characteristic of our evaluation method: Depending on the preferences of the cloud provider, there may exist more than one optimal choice how to configure a continuous test. Through adding further constraints to our assumptions about the provider's preferences, we may be able to decrease the number of optimal solutions, possibly to a single one. Yet there are other properties of a test which can be considered in order to guide selection of the most suitable one, e.g., taking into account the overhead incurred by the cloud service under test.

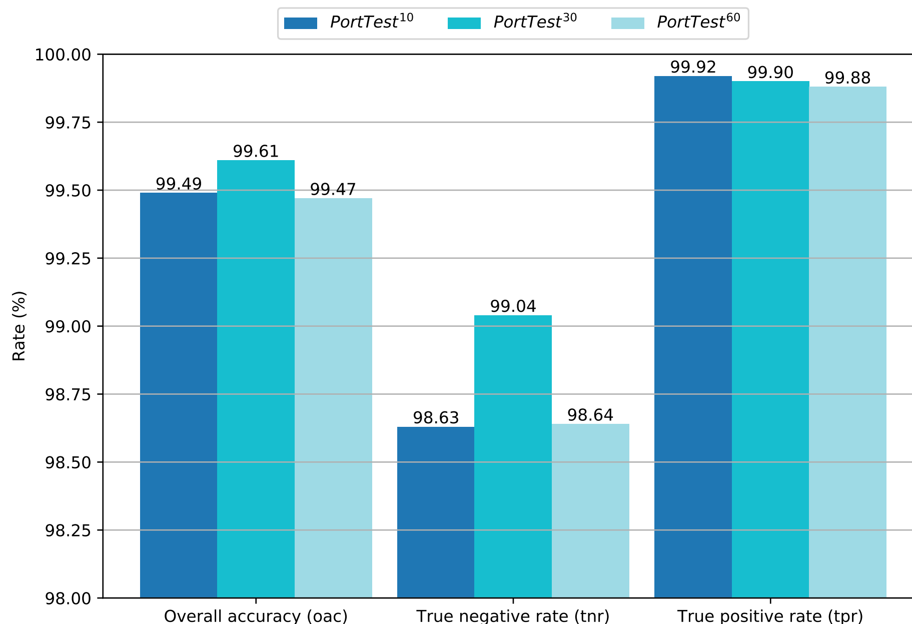


Figure 7.23: Selected accuracy measures for PortTest^{10}, PortTest^{30} and PortTest^{60} based on the Basic-Result-Counter (brC) test metric

Table 7.15: Evaluation of PortTest to test secure interface configuration of $PaaS^{OS}$ based on the Basic-Result-Counter (brC) test metric

Accuracy & precision measures		PortTest		
		10	30	60
ebrC (%)	oac	99.49	99.61	99.47
	$E^{95\%}_{oac}$	0.09	0.14	0.23
	tnr	98.63	99.04	98.64
	$E^{95\%}_{tnr}$	0.26	0.38	0.64
	tpr	99.92	99.90	99.88
	$E^{95\%}_{tpr}$	0.04	0.09	0.13
	fnr	0.15	0.20	0.24
	$E^{95\%}_{fnr}$	0.09	0.18	0.27
	fpr	1.37	0.96	1.36
	$E^{95\%}_{fpr}$	0.26	0.38	0.64
	fdr	0.69	0.48	0.67
	$E^{95\%}_{fdr}$	0.13	0.19	0.32
	ppv	99.31	99.52	99.33
	$E^{95\%}_{ppv}$	0.13	0.19	0.32
	for	0.15	0.20	0.24
	$E^{95\%}_{for}$	0.09	0.18	0.27
	npv	99.85	99.80	99.76
	$E^{95\%}_{npv}$	0.09	0.18	0.27

Table 7.16: Evaluation of PortTest to test secure interface configuration of $PaaS^{OS}$ based on the Failed-Passed-Sequence-Counter ($fpsC$) test metric

Accuracy & precision measures		PortTest		
		10	30	60
efpsC (%)	tnr	99.9	99.9	99.8
	$E^{95\%}_{tnr}$	0.20	0.20	0.28
	fpr	0.10	0.10	0.20
	$E^{95\%}_{fpr}$	0.20	0.20	0.28
	for	1.50	0.50	0.30
	$E^{95\%}_{for}$	0.75	0.44	0.34
	npv	98.50	99.50	99.70
	$E^{95\%}_{npv}$	0.75	0.44	0.34

Table 7.17: Evaluation of PortTest to test secure interface configuration of $PaaS^{OS}$ based on the Failed-Passed-Sequence-Duration ($fpsD$) test metric

Accuracy & precision measures		PortTest		
		10	**30**	**60**
$efpsD^{TN}$ (ms)	$\bar{x}$	-628	-330	-804
	$\widehat{x}$	-357.5	308	-9051
	sd	5149	11681	24102
	min	-64026	-37171	-30158
	max	9625	27583	59143
	$E^{95\%}$	321	727	1499
$efpsD^{TN}_{rel}$ (%)	$\bar{x}$	4.87	12.27	24.96
	$\widehat{x}$	3.97	10.48	22.1
	sd	4.95	9.66	18.11
	min	0.006	0.003	0.009
	max	86.23	55.24	96.58
	$E^{95\%}$	0.31	0.60	1.13
$efpsD^{TN}_{pre}$ (ms)	$\bar{x}$	6020	15517	31097
	$\widehat{x}$	5883	15419	30726
	sd	4293	8895	17431
	min	280	660	731
	max	73046	49769	60888
	$E^{95\%}$	268	554	1084
$efpsD^{TN}_{post}$ (ms)	$\bar{x}$	5391	15187	30294
	$\widehat{x}$	5407	15374.5	30052
	sd	2954	8547	17381
	min	110	147	314
	max	11072	30120	60698
	$E^{95\%}$	184	532	1081
$efpsD^{FN}$ (ms)	$\bar{x}$	67026	90818	120385
	$\widehat{x}$	72673	90906	120272
	sd	24370	219	241
	min	10247	90428	120221
	max	94479	90935	120662
	$E^{95\%}$	13496	272	599
$efpsD^{FP}$ (ms)	$\bar{x}$	*na*	*na*	66252
	$\widehat{x}$	*na*	*na*	66252
	sd	*na*	*na*	5657
	min	61167	76256	62252
	max	61167	76256	70252
	$E^{95\%}$	*na*	*na*	50825

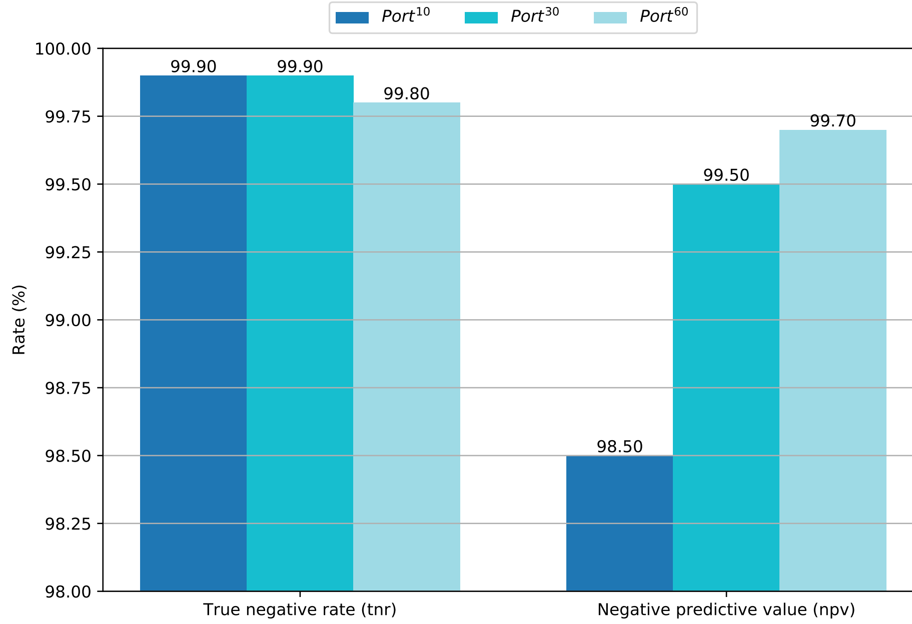


Figure 7.24: Selected accuracy measures for PortTest^{10}, PortTest^{30} and PortTest^{60} based on the Failed-Passed-Sequence-Counter ($fpsC$) test metric

Table 7.18: Evaluation of PortTest to test secure interface configuration of $PaaS^{OS}$ based on the Cumulative-Failed-Passed-Sequence-Duration ($cfpsD$) test metric

Accuracy & precision measures		**PortTest**		
		10	**30**	**60**
$ecfpsD^{TN}$	TN(ms)	-1493904	-781150	-1162889
	TN(%)	1.97	1.04	1.55
$ecfpsD^{FN}$	FN(ms)	1005404	454092	361155
	FN(%)	1.33	0.6	0.49
$ecfpsD^{FP}$	FP(ms)	61167	76256	132504
	FP(%)	0.08	0.10	0.18

7.6 Summary and discussion

In this chapter, we presented a method to experimentally evaluate the accuracy and precision of tests supporting continuous test-based cloud certification. To that end, we used the universal test metrics brC, $fpsC$, $fpsD$ and $cfpsD$ introduced in Section 4.3.5 to define what accuracy and precision mean in context with continuous test-based certification. Then we introduced a process which consists of all steps necessary to experimentally evaluate the results produced by a continuous test. At this point, we presented the notion of control violation sequences consisting of control violation events during which a property of a cloud

service under test is manipulated such that one or more controls of a certificate are violated. Thus control violation sequences establish the ground truth which we then use to evaluate the accuracy and precision of a continuous test. Further, we explained how randomizing duration of as well as interval between control violation events allow for general statements – in the limits of the range from which random values are selected – about a test's accuracy.

At the core of our method are accuracy and precision measures which are based on the set of universal test metrics, i.e., brC, $fpsC$, $fpsD$ and $cfpsD$ (see Section 4.3.5). We derive these measures by combining the notion of control violation sequences with our definition of continuous test's accuracy in context of the universal test metrics. These measures allow us to make statements about, e.g., the average error a particular test makes when estimating the duration of a control violation event.

Finally, we presented three example scenarios to show how the accuracy and precision measures allow us to compare alternative tests as well as alternative test configurations to select the one most suited. In this context, we highlight an important limitation of our approach: Our method only reveals but does not solve disagreement between results of accuracy and precision measures. Consequently, it depends on the details of the use case which test or test configuration is the preferable one. However, developing a scenario driven approach which allows to select the accuracy and precision measure most suited for a particular use case is outside the scope of this work.

As already pointed out above, the evaluation approach presented in this chapter is based on the set of universal test metrics which, in turn, are conceived as part of the framework in Chapter 4. This implies that evaluations are only applicable to such test designs which implement these test metrics. It is important to note that the set of evaluation measures proposed in this chapter is not complete and, furthermore, there are numerous other feasible test metric definitions from which accuracy and precision measures can be derived. However, as of writing this thesis, no other approaches have been published which allow to evaluate accuracy and precision of test-based evidence production techniques. It is therefore left to future work to build on the set of evaluation measures proposed herein and extend this set as required, e.g., by introducing scenario specific test metrics and derive corresponding evaluation measures.

Another deficit of our approach consists of the somewhat arbitrary selection of duration and frequency of control violation events. Naturally, the most desired distributions to use when triggering control violation events are those observed in the wild, e.g., a historic sequence of downtimes AWS had during the past year. However, obtaining such fine-grained documentation about count and duration of events where a cloud provider does not comply with certificate's controls – if even existent – is obviously hard or impossible. Naturally, continuous tests themselves are a means to collect such data. Yet this implies that we would then accept this data to establish the ground truth. Since our method to evaluate the accuracy and precision of continuous test aims at reasoning *about* these continuous test results, the results produced may be erroneous, leading to incorrect assumptions about the observed distributions of control violation events. A more promising approach appears to be to define different types of probability distributions for control violation events and conduct separate evaluations of a continuous test. That way, statements about the accuracy and precision of a test in context with a particular type of control violation sequence can be made, e.g., how well does a particular test work when trying to detect rare, short-lived control violations?

Finally, the accuracy and precision of a continuous test may not be the only two characteristics which are needed to determine the most suitable test in a real world scenario.

Another important property of a continuous test is the overhead which it imposes on the service under test, especially when facing multiple, possibly concurrent tests. An interesting idea here is to measure the overhead caused by tests and then select those tests and configurations which incur minimal overhead while retaining required accuracy and precision of results.

Chapter 8

Test-based certification of opportunistic cloud service providers

In the previous chapters, we introduced a framework to design tests supporting continuous cloud service certification (Chapter 4), presented example test scenarios (Chapter 5), showed how to rigorously define continuous tests (Chapter 6) and proposed a method to evaluate the accuracy and precision of continuous test results (Chapter 7). Up to this point, we implicitly assumed that we can trust the cloud service provider, that is, we presumed that our produced test results are trustworthy. In this section, we address *Research Challenge 4: Trustworthy continuous test results* described in Section 1.2.4 by relaxing this assumption through introducing the notion of *opportunistic cloud service providers*.

Recall that through certifying their cloud services, cloud providers aim to demonstrate compliance with standards and guidelines such as CSA's Cloud Control Matrix [22], BSI C5 [31] or ISO/IEC 27001:2013 [24]. The cloud provider's expectations which motivate certification are competitive advantages in the form of, e.g., increased customer retention and attraction of new service customers. While such competitive advantages may be hard to quantify, satisfaction of controls necessary to obtain a particular certificate can increase costs of the provider. In order to reduce costs of certification, the *opportunistic provider* only pretends to satisfy controls. However, although fraudulent, opportunistic providers do not cheat arbitrarily, on the contrary: They only cheat if they can assume with (some) certainty that they are not caught. The intuitive reason behind that is that getting caught when deceiving, the provider incurs some kind of intolerable penalty, e.g., loss of cloud service customers and damage to the provider's reputation.

Yet not trusting the cloud service provider means that we can no longer unconditionally trust test results. Therefore, we have to adapt our tests in order to produce trustworthy results. To that end, in this chapter, we introduce an approach to model the behavior of an opportunistic cloud service provider which combines the notion of *covert adversaries* introduced by Aumann and Lindell [44] in the context of secure multiparty computation with *labeled transition systems (LTS)*, a formal method to describe the behavior of systems. Using this model, we can reason about the behavior of opportunistic cloud providers and show how randomization of tests can reduce their willingness to cheat.

The next section outlines the main concepts of covert adversaries and labeled transition systems. Thereafter, we describe how we use LTS to model opportunistic cloud service providers (Section 8.2). Finally, we show how randomization of tests can reduce the willingness of opportunistic providers to deceive and point out how our framework to design

tests presented in Chapter 4 supports randomization (Section 8.3.2). Parts of this chapter have been published in [244].

8.1 Background

This section outlines the main ideas behind covert adversaries and labeled transition systems.

8.1.1 Covert adversaries

Covert adversaries are a special type of adversaries which were introduced by Aumann and Lindell [44] in the context of secure multiparty computation. Those adversaries are willing to actively cheat during execution of a multiparty computation protocol but only if they do not get caught when deceiving.

As early as 1982, Yao [302] first described the problem of multiparty computation using an introductory example which is now referred to as the *Yao's Millionaires' Problem*: Two millionaires seek to find out who is richer without revealing to each other their actual wealth. Yoa proved that a set of n parties can efficiently compute the output of a n-input function where every party obtains the correct output but no further information which the participating parties did not possess prior to the computation. In context of *Yao's Millionaires' Problem*, this means that there exists a function to which the two millionaires provide their secret amount of wealth as input and the function outputs who is indeed richer (but not more).

Informally, multiparty party computation problems have to satisfy the following two requirements [303]:

1. A participating party only learns the results of the computation, and
2. the output of the computation is correct.

An obvious question at this point is what happens if one or more parties are adversarial. In order to analyze security properties of multiparty computation, Aumann and Lindell distinguish three types of adversaries [44]:

1. *Malicious adversaries:* This type of adversary behaves arbitrarily, that is, malicious adversaries are not constrained by the protocol specified to conduct multiparty computation. Therefore, if a multiparty computation protocol is secure against malicious adversaries, then these protocols provide honest participating parties with the guarantee that no information other than the computation's result can be obtained by the parties.
2. *Semi-honest adversaries:* In contrast to malicious adversaries, semi-honest adversaries are bound by the protocol which is specified for multiparty computation. However, these adversaries try to infer additional information from any steps taken during the execution of the computation. Therefore, a multiparty computation protocol which is secure against semi-honest adversaries guarantees honest parties that no information is inadvertently leaked during protocol execution. Compared to being secure against malicious adversaries, this security guarantee is rather weak because it implies that participating parties trust each other.

3. *Covert adversaries:* This type of adversary lies between the malicious and the semi-honest adversaries: They are willing to actively cheat and thus are not semi-honest while, at the same time, they will only do so if they are not caught, thereby rendering them non-arbitrary. Covert adversaries do not necessarily avoid being caught deceiving at any price, that is, the presumed advantage from cheating may outweigh the penalty of getting caught. Therefore, if a multiparty computation protocol is secure against covert adversaries, then it guarantees the honest participating parties that there is only negligible incentive for a covert adversary to cheat.

Aumann and Lindell build on the *ideal/real simulation paradigm* to define covert adversarial behavior [44]. The basic notion of this paradigm is to define an *ideal* model of the protocol which is secure by definition and then compare this ideal model to an adversaries' *real* capabilities. The ideal model uses the fiction that a trusted third party exists to whom the participating parties send their inputs and which then – after having computed the result – provides the output to the parties. Based on this idea, a *real* protocol is considered secure if it satisfies the ideal model's properties, that is, if an adversary's capabilities in the real model are identical with those in the ideal model. In context of defining covert adversarial behavior, one possibility is to treat successfully deceiving as a kind of behavior which is not possible within the ideal model of the protocol.

Any attempt of a covert adversary to cheat is detected with a probability of at least $\epsilon \in (0, 1]$ to which Aumann and Lindell refer as the *deterrence factor*. Therefore, if an adversary has identified a deceptive behavior which is expected to be successful with a probability of p, then the honest participating parties will catch the cheating adversary with at least $p \times \epsilon$. The greater the value of ϵ, the greater the probability that a covert adversary gets caught cheating. Hence the security of the protocol is described with *security in the presence of covert adversaries with ϵ-deterrent.* It is important to note that a covert adversary getting caught cheating does not necessarily mean that his deception is not successful.

Covert adversaries can have different characteristics. Hereafter, the main three variants present by Aumann and Lindell [44] are outlined.

1. *Failed-simulation formulation:* If an adversary cannot successfully cheat, then the execution of the ideal model and the real model of the protocol are indistinguishable. In contrast, the *failed-simulation* definition of a covert adversary assumes that comparing the ideal model of the protocol with the real execution can sometimes be distinguishable, i.e., the ideal model of the protocol execution *fails*. Such cases are simply events of successful cheating. Put differently: There is only a probability Δ that the output distributions of the ideal and of the real protocol execution are distinguishable. This means that there is a probability of at least $\Delta \times \epsilon$ that honest participating parties catch a covert adversary when deceiving.

2. *Explicit-cheat formulation:* In this formulation, the covert adversary is given *explicit* means to cheat. To that end, the adversary can instruct the trusted third party – which is part of the ideal model definition – to supply it all secret inputs which the other participating parties have provided. Then the trusted third party decides with a probability ϵ whether it informs the honest participating parties that cheating has occurred. Thus the covert adversary can always decide to cheat but has to expect to get caught with a probability of ϵ. However, since the ideal model of the protocol has been extended through the notion of the explicit cheat instruction, the output distribution of the ideal model is indistinguishable from the real execution of the protocol.

3. *Strong explicit-cheat formulation:* This third definition modifies the explicit-cheat formulation in the following way: The covert adversary only obtains the secret input of any other participating party if the trusted third party does *not* disclose that cheating has taken place. The formulation is *stronger* than the previous one because a covert adversary who gets caught cheating gains no information about the inputs of the others.

In order to model an opportunistic cloud provider, we will build on the notion of the *explicit-cheat formulation*. This will be explained in detail in Section 8.2.1.3.

8.1.2 Labeled Transition Systems

Labeled transition systems (LTS) are a class of models which are used to describe the behavior of a system. A comprehensive introduction to LTS can be found in, e.g., Baier and Katoen [304] which depicts the primary source we use to describe the basic concepts of LTS in this section.

LTS are applied within *model checking*, an approach which uses formal methods to establish the correctness of a system in a mathematically precise and unambiguous manner. The mathematical structure which is used to define LTS are directed graphs. The nodes of the graph correspond to the *states* of the LTS while the edges model *transitions* between the states. A *state* captures information at a chosen level of abstraction of the system which is described by the LTS. Thus, a sequence of states describes how the system evolves where changes from one state to another are specified by *transitions*.

Before formally defining a LTS, there are two further concepts which need to be introduced. The first one are referred to as *actions* which are simply names for the transitions. Actions are needed to model communication mechanisms between processes which is not required for our purposes. In cases where action names are not relevant because they only refer to internal processes, the special symbol τ is used to indicate a placeholder. The second one are *atomic propositions* which express some known facts about a state at some point in time.

Definition Based on the above concepts, we can define a labeled transition system (LTS) as a 6-tuple $\langle S, Act, \longrightarrow, I, AP, L\rangle$:

- the set of states S,
- the set of Actions Act assigning names to transactions of the LTS,
- the transition relation $\longrightarrow \subseteq S \times A \times S$
- the set of initial states I where the LTS can start from,
- the set of atomic propositions AP representing a state's temporal properties, and
- a labeling function $L : S \rightarrow 2^{AP}$ assigning each state a subset of atomic propositions.

Behavior An LTS starts at an initial state $s_0 \in I$ and evolves as specified by the transition relation $\longrightarrow$, that is, $(s_0, \alpha, \hat{s}) \in \longrightarrow$. More generally, if a LTS is at some state s, then a transition to a next state $\hat{s}$ is selected *nondeterministically* and the action α is performed. This selection repeats until the LTS has evolved to a state which has no outgoing transition. At each state the labeling function is evaluated. To that end, L takes a state $s \in S$ as input and outputs the atomic propositions for that state, i.e., $L(s) \in 2^{AP}$.

Note that, loosely speaking, *nondeterministic selection* of outgoing transitions means that there is no information which outgoing transition is selected if a state has more than one. This nondeterminism comes in handy when a certain level of abstraction on the model is needed, e.g., when modeling interfaces with unknown environments, such as user interfaces where no information about a user's choice is available.

Yet this notion of nondeterminism becomes somewhat difficult if it is required to observe the behavior of a LTS. To that end, observable behavior of a LTS has to be defined for which two general approaches exist: The first one is referred to as *action-based* and defines that only executed actions can be observed. Such LTS are called *action-deterministic* and have at most one outgoing transition $\alpha \in A$ per state. The second approach is called *state-based* and defines that only satisfied atomic propositions AP of a state can be observed, neglecting the actions' labels. Such LTS are referred to as *AP-deterministic* and have at most one outgoing transition which leads to a state with label $l \in AP$.

A LTS formalizes the behavior of a system which it describes using *executions* or *runs*. Executions resolve the nondeterminism which was discussed above by explicitly describing one possible behavior of the LTS. Neglecting the actions of a LTS, one possible behavior of an LTS is captured by a sequence of states

$$\rho = \langle s_0, s_1, \ldots s_n \rangle.$$

Note that the above execution is finite since it contains n elements. We omit the discussion of *infinite* execution of a LTS because it is not required to model opportunistic cloud service providers. The following section will explain in detail how we use LTS to describe the behavior of an opportunistic provider.

8.2 Opportunistic provider as a labeled transition system

In this section, we describe how we use a labeled transition system (LTS) as a tool to describe the behavior of an opportunistic cloud provider. It is important to note that we focus on describing a provider who *intentionally* does not satisfy controls of a certificate. The case that a provider *incidentally* does not comply with the controls, e.g., as a result of safety issues, is outside of the scope of this thesis.

We define the opportunistic provider as the labeled transition system OCP which is a 6-tuple consisting of the set of states S, the set of actions A, the transition relation $\longrightarrow \subseteq S \times A \times S$, the set of initial states I, the set of atomic propositions AP, and a labeling function $L : S \rightarrow 2^{AP}$:

$$OCP = \langle S, A, \longrightarrow, I, AP, L \rangle.$$

As pointed out in Section 8.1.2, the selection of transition in OCP is nondeterministic, that is, if a state has multiple outgoing transitions, then it is not known beforehand which of the available transitions will be selected. In order to be able to observe the behavior of an opportunistic cloud provider, we adopt the *state-based* approach. This means that we assume that we can observe atomic propositions of a state of OCP which hold. Put differently: We assume that we can observe states' labels through testing. Lastly, this *AP-deterministic* view of OCP implies that we ignore the actions which lead to state changes.

In the following sections, we explain the states, actions, atomic propositions and the labeling function of OCP. Note that we omit the definition of transition relation $\longrightarrow$ as well

as the initial states I. In the case of the transition relation, knowing internal actions of OCP would be required to provide a meaningful description of $\longrightarrow$ because each element of $\longrightarrow$ is a 3-tuple $(s, \alpha, \hat{s})$. However, as we will explain in Section 8.2.3, we do not assume to be able to observe actions of OCP.

8.2.1 States

A state $s \in S$ of the labeled transition system OCP is defined by evaluating the three variables: Unique identity of a state (u), cost of a state (c), and deception of a provider at a state (d). These three variables are explained hereafter.

8.2.1.1 Unique identity

As the name suggests, the variable u is used to assign a unique identity to a particular state of the OCP. Although this variable is not essential to describe the behavior of an opportunistic cloud provider, it serves as a convenience to describe specific states of an OCP. Examples for such identities are names which describe the current state of a cloud service, e.g., $u =$ *vm1_is_running* or $u =$ *network3_is_disconnected*.

8.2.1.2 Cost

In order to provide a cloud service customer with a cloud service, a cloud provider has to allocate some resource. This allocation of resources is usually expressed as a quantitative value, i.e., costs. Consider, for example, a cloud service customer has triggered the start of a virtual machine. The provider will then identify and select a host with sufficient resources, then initialize boot of the image selected by the customer and configure the required network services and so forth. Other costs which the provider incurs are caused by operating the cloud service, for example, running diagnostics of the underlying network and host infrastructure as well as reserving hot spares in case some components involved in delivery of the cloud service fail.

In order to model the costs which a cloud provider incurs through service provisioning, we have to distinguish between two types of cost: The first one are so-called *local costs* which are costs $c \in \mathbb{Q}^+$ that are assigned to each state $s \in S$. Note that we assume that the cloud provider always incurs costs when supplying a service to a customer, that is, $c > 0$. Further, we omit defining a cost function which evaluates the costs of a specific state of an OCP because such costs are obviously highly dependent on the specific scenario, e.g., the cloud service which is provided.

The second type of cost are referred to as *accumulated costs* which are costs that a cloud provider incurs through his behavior. The *behavior* of a cloud service provider is the sequence of states

$$\upsilon_{i,k} = \langle s_i, s_{i+1}, \dots, s_k \rangle \in \Upsilon_{i,k}$$

where $\Upsilon_{i,k}$ delineates the set of all possible sequences that exist starting from state s_i and ending at state s_k.

We can now combine the idea of local and accumulated costs to describe the *costs of a cloud service provider's behavior*. To that end, we define the function $B : \Upsilon \rightarrow \mathbb{Q}^+$ which

takes as input sequence of states $\upsilon \in \Upsilon$ and outputs the costs of that sequence by summing over all costs c of $s \in \upsilon_{i,k}$:

$$B(\upsilon_{i,k}) = \sum_{s=i}^{k} c_s.$$

8.2.1.3 Deception

The state of a cloud provider satisfies the controls of a certificate if all controls hold at that state. Yet if a cloud service provider cheats at any state, then at least one of the certificate's controls is dissatisfied at that state. If a cloud provider does not satisfy all the controls of the certificate, then the cloud service is not compliant.

At this point, it is important to note that this is a technical detail because the satisfaction of a certificate's controls are not restricted to a single state of OCP. Rather, we want to describe that a certificate's controls are satisfied by the behavior of a cloud provider, i.e., by a sequence of states $\upsilon_{i,k}$. In this context, controls of a certificate may even tolerate some non-compliant states within the sequence. Consider, as an example, the cloud provider guarantees that *the cloud service is available at least 95% per year*. Such a guarantee depicts one possible operationalization of, e.g., control *IVS-04* of the CSA's CCM [22] (see Section 5.2.2.1 for further detail); therefore, violating this service level agreement leads to violation of control *IVS-04*. However, the agreement – and thus the control – has a 5% limit of tolerance up to which the behavior of the cloud provider, that is, making the cloud service available is allowed not be satisfied.

The variable $d \in \{0, 1\}$ indicates whether the cloud provider is cheating at state s. It is important to point out that we assume that testing the cloud provider always returns correct results. This means that if we test a cloud provider at a state s where he is *not* compliant because he is cheating, i.e., $d_s = 1$, then our tests will indicate that a control is not satisfied. The implications and analysis of inaccurate test results have been thoroughly studied in Chapter 7.

Since certification of cloud services is a voluntary act, we assume that the cloud provider knows that he is subject to test-based certification. In order to denote whether a cloud provider's compliance with certificate's controls is tested at a state s, we define $\eta \in \{0, 1\}$. If $\eta = 1$, then the provider is tested, and if $\eta = 0$, then the provider is not tested. Now we have to describe the knowledge a cloud provider has about whether he is tested at a state s. To that end, we use the probability $P_s(\eta)$: If the cloud provider is tested at state s and he knows that he is tested at that state, then $P_s(\eta = 1) = 1$. If, in turn, the provider is *not* tested at state s and knows this, then $P_s(\eta = 0) = 1$.

This probabilistic approach to describe a cloud provider's knowledge about whether he is tested at a state draws on the notion of the explicit-cheat formulation of a covert adversary introduced in Section 8.1.1. Recall that the covert adversary which explicitly cheats depends on a trusted third party to disclose his deceit with a probability of ϵ. Similarly, if an opportunistic cloud provider decides to cheat at state s, then there is a probability of $P_s(\eta = 1)$ that he is caught cheating.

Now we can combine the ideas presented above to derive the *probability of detecting a provider's deceptive behavior*. Recall that a provider cheating at a state s is described by the variable d. Furthermore, the probability that a provider's compliance is tested at state s is defined by $P_s(\eta = 1)$. We define the function $D : \Upsilon \rightarrow [0, 1]$ to describe the probability of

detecting a cloud provider when deceiving. This function takes as input the behavior of the cloud provider, i.e., a sequence of states $\upsilon_{i,k} \in \Upsilon_{i,k}$ and maps it to the interval [0, 1]:

$$D(\upsilon_{i,k}) = \frac{\sum_{s=i}^{k} P_s(\eta = 1) \times d_s}{|\upsilon_{i,k}|} \text{ with } i, k \in \mathbb{N} \tag{8.1}$$

This function can interpreted as follows: The probability of detecting deceptive behavior of the cloud provider OCP increases the more states the cloud provider deceives at, i.e., $d_s = 1$, and also presumes that these states are subject to testing, i.e., $P_s(\eta = 1) > 0$.

From the opportunistic cloud provider's point of view, deceptive behavior is *successful* if one or more states s of his behavior, that is, $\upsilon_{i,k}$ do not comply with the controls of the certificate, i.e., $d = 1$, while the probability D of getting caught while deceiving is less than or equal to his *opportunistic limit* $\varphi \in [0, 1)$. The opportunistic limit delineates the maximum probability of getting caught when cheating that an opportunistic cloud provider is willing to accept. Observe that the bounds of φ exclude 1, that is, $0 \leq \varphi < 1$ because otherwise a cloud provider may not be opportunistic but accepts certain detection when deceiving.

8.2.2 Atomic propositions and labeling function

As mentioned above, we choose the state-based approach (or AP-deterministic LTS) to be able to observe the behavior of OCP. Therefore, when testing, we are only able to observe whether the atomic propositions of a state of OCP hold. Furthermore, we assume that we cannot observe the evaluation of the costs c and the unique id u through testing. These assumptions appear to be reasonable because evaluating the costs at some state s involves a multitude of other variables such as labor costs of administrators which are hardly observable through testing the cloud service of the cloud provider.

What we can observe through testing, however, is whether a cloud provider complies with the controls of a certificate at a state s. This is captured by a state's variable d. Therefore, we define the labeling function L which takes d of a state s as an input and outputs a label for s indicating whether a provider's state is $\{compliant\}$ or $\{not_compliant\}$ with controls of the certificate:

$$L(d) = \begin{cases} \{compliant\} & \text{if } d = 0 \\ \{not_compliant\} & \text{if } d = 1. \end{cases} \tag{8.2}$$

8.2.3 Actions

As pointed out in the previous section, we adopt the state-based approach to observe the behavior of the opportunistic cloud provider OCP. This implies that we cannot observe the actions of OCP through testing. Therefore, we use τ to label a provider's actions $A \in OCP$. These actions consist of any action a cloud provider needs to take in order to provide cloud services to cloud service customers. This includes launching new services instances, managing resource allocation, and enforcing security settings.

In this context, it is important to note that we do not restrict the capabilities of a cloud provider within his domain, that is, the provider may access and modify any component which is involved in delivery of his cloud services. These capabilities allow the provider to read, insert, modify or delete any application data and network traffic with his infrastructure.

8.2.4 Choice of behavior

After having defined the opportunistic cloud service provider as the labeled transition system OCP, this section describes how the costs of a provider's behavior and the probability of detecting deceptive behavior are related. We can use this relationship to reason about the behavior an opportunistic cloud provider prefers.

8.2.4.1 Optimal behavior

Recall that the behavior of a cloud provider is modeled as the sequence of states $\upsilon_{i,k}$ which starts at state s_i and ends at state s_k. Thus, alternative behavior of the cloud provider translates to additional or alternative states, thereby altering the sequence of states $\upsilon_{i,k}$ to $\widehat{\upsilon}_{i,k}$. As a result, the alternative behavior may change accumulated costs from $z_{i,k}$ to $\widehat{z}_{i,k}$.

The opportunistic provider prefers to deceive, that is, not to comply with a certificate's controls if accumulated costs $\widehat{z}_{i,k}$ of successfully deceiving, i.e., $D(\widehat{\upsilon}_{i,k}) \leq \varphi$ are lower than behaving to the satisfaction of a certificate's controls. More formally, if the opportunistic provider has to choose between behavior alternatives, then this choice can be formulated as the following optimization problem:

$$\begin{aligned} \underset{\upsilon_{i,k} \in \Upsilon_{i,k}}{\text{minimize}} \quad & B(\upsilon_{i,k}) \\ \text{subject to} \quad & D(\upsilon_{i,k}) \leq \varphi \end{aligned} \tag{8.3}$$

The selection process can be described as follows: First, the opportunistic provider discovers all available behavior alternatives which start at state s_i and end at state s_k, that is, $\upsilon_{i,k} = \langle s_i, s_{i+1} \ldots s_k \rangle \in \Upsilon_{i,k}$. Thereafter, he calculates the costs B as well as the probability of detection D for all behavior alternatives $\upsilon_{i,k} \in \Upsilon_{i,k}$. As the final step, the provider then chooses the behavior, that is, the sequence of states $\upsilon_{i,k}$ where the costs B are minimal under the constraint that the probability of deception D is lower or equal to the opportunistic limit φ.

Note that there exist a corner case where the chosen behavior may satisfy the opportunistic limit but is *prohibitively expensive*. In this case, the provider will prefer to cheat even though the probability of getting caught exceeds his tolerance. Put differently: It is cheaper to get caught when cheating than it is to comply with a certificate's controls. In order to capture this case, we introduce an upper bound $\omega \in \mathbb{Q}^+$ to the cost of a chosen behavior. A behavior is prohibitively expensive if $B(\upsilon_{i,k}) > \omega$.

An important question at this point is how to define this upper bound ω in a concrete scenario. This implies having a robust method at hand to quantify the cost of non-compliance for the cloud provider, e.g., through loosing service customers or damages to the provider's reputation. We note that – although a justified question – it is outside the scope of this thesis to examine the effects of non-compliance in a specific setting and the resulting loss of certification of cloud services on the cloud provider.

8.2.4.2 Example of an opportunistic cloud provider

The following example illustrates how an opportunistic cloud provider OCP chooses to behave. In our example, this provider offers IaaS to service customers, that is, virtual machines (VM). According to the certificate which these IaaS possesses, the provider is required to ensure that each VM is available 99% per year which translates to $0.01 \times 365 \times 24 \approx 87$ hours of downtime in a year of 365 days.

Figure 8.1 shows an instance of this OCP which we refer to as OCP^{VM} consisting of the three following states:

$$S = \{\{u = u_1 = start^{VM}, c = 2, d = 0\}, \{u = u_2 = run^{VM}, c = 10, d = 0\}, \{u = u_3 = stop^{VM}, c = 5, d = 1\}\}.$$

In order to keep this example simple, we assume that through running the VM, the cloud provider incurs a cost of 10 per hour. The hourly reoccurring cost is modeled by OCP^{VM} traversing from u_2 to u_2 every hour. Analogous to incurring costs by running a VM, stopping a VM also incurs an hourly cost, i.e., remaining in state u_3 incurs a cost of 5 every hour. Complying with the certificate's availability control means that the provider has to keep the

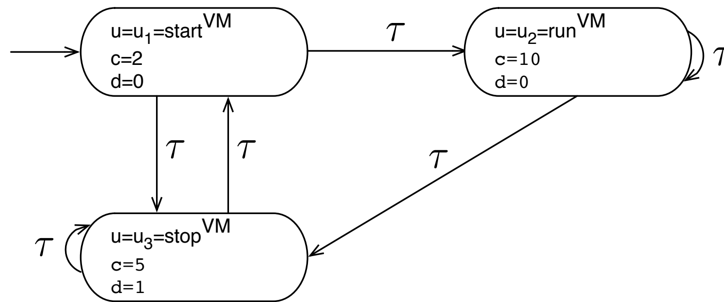


Figure 8.1: Example LTS OCP^{VM} consisting of three states

VM in state u_2 for at least $365 \times 24 - 87 = 8673$ hours per year. Therefore, one behavior, that is, a sequence of states which complies with the certificate's control is $\upsilon_1 = \langle u_1, 8673 * u_2, 87 * u_3 \rangle$. Behaving according to υ_1 means that the provider starts the VM (u_1), runs it for a total of 8673 hours in a row ($8673 * u_2$) and then stops it for the remaining hours of the year ($87 * u_3$). The accumulated costs of this behavior is

$$B(\upsilon_1) = 1 \times 2 + 8673 \times 10 + 87 \times 5 = 87167.$$

Lets now assume that the provider knows that he is test at each state, that is,

$$\forall s \in \upsilon_{i,k} : P_s(\eta = 1) = 1.$$

Recall the above scenario where the provider stops the VM for a total of 87 hours. The detection probability of this behavior υ_1 is therefore

$$D(\upsilon_1) = \frac{\sum_{s=i}^{8761} P_s(\eta=1) \times d_s}{|\upsilon_1|} = \frac{(1\times0+87\times1+8673\times0)}{8761} \approx 0.01.$$

It is important to note that this behavior of the cloud provider actually satisfies the certificate's control related to availability although some states $s \in \upsilon_1$ do not comply. This example can be interpreted as a case where the opportunistic cloud provider configures his opportunistic limit according to the tolerance permitted by the control of the certificate.

In the following two examples, we will consider two behavior alternatives which do not comply with the availability control of the certificate. In the first example, the cloud provider starts the VM, runs it for a total of 8000 successive hours and stopping it for 760 hours, that is, $\upsilon_2 = \langle u_1, 8000 * u_2, 760 * u_3 \rangle$. This incurs a total cost of

$$B(\upsilon_2) = 1 \times 2 + 8000 \times 10 + 760 \times 5 = 83802.$$

and has a detection probability of

$$D(\upsilon_2) = \frac{\sum_{s=i}^{8761} P_s(\eta=1)\times d_s}{|\upsilon_1|} = \frac{(1\times 0+760\times 1+8000\times 0)}{8761} \approx 0.087.$$

In the second example, we assume that the cloud provider only runs the VM for 6000 hours, leaving 2760 hours where the VM is stopped. This incurs a cost of

$$B(\upsilon_3) = 1\times 2 + 6000\times 10 + 2760\times 5 = 73802$$

with a detection probability of

$$D(\upsilon_3) = \frac{\sum_{s=i}^{8761} P_s(\eta=1)\times d_s}{|\upsilon_1|} = \frac{(1\times 0+2760\times 1+6000\times 0)}{8761} \approx 0.315.$$

Let's assume that OCP^{VM} has an opportunistic limit of $\varphi_{ocp} = 0.1$ and can choose either behavior υ_1, υ_2 or υ_3. In this scenario, the provider will prefer υ_2 because it is less expensive than υ_1 and detection probability is lower than his opportunistic limit φ_{ocp}.

8.3 Randomization of tests as a countermeasure

In this section, we start with showing how randomization can reduce the willingness of an opportunistic cloud service provider to cheat. Thereafter, we describe how our framework to design tests supporting continuous cloud service certification (see Chapter 4) supports randomization (Section 8.3.2).

8.3.1 Effect of randomization on opportunistic behavior

Lets first consider an opportunistic cloud provider who has complete knowledge about tests, that is, knows for certain which states are tested and which are not. If this provider does not want to get caught cheating at all, i.e., his opportunistic limit is 0, then he will select a behavior $\upsilon_{i,k}$ which minimizes his accumulated costs while only deceiving at states at which he knows he is not tested.

Having described this intuition about an opportunistic cloud provider with complete knowledge, we will now formalize this notion. Recall that a behavior of the opportunistic provider is described by the sequence $\upsilon_{i,k}$. This sequence holds in total $|\upsilon_{i,k}|$ elements. Let a be the number of states in $\upsilon_{i,k}$ at which the provider deceives and let b be the number of states where he satisfies a certificate's controls. The sum of a and b equal the number of elements of $\upsilon_{i,k}$, i.e., $a + b = |\upsilon_{i,k}|$. We can now use a and b to rewrite our definition of the probability of detecting a provider's deceptive behavior D which we introduced with Equation 8.1 in Section 8.2.1 as follows:

$$\begin{aligned} D(\upsilon_{i,k}) &= \frac{\sum_{s=i}^{k} P_s(\eta = 1)\times d_s}{|\upsilon_{i,k}|} \\ &= \underbrace{\frac{\sum_{s=1}^{a} P_s(\eta = 1)\times d_s}{(a+b)}}_{=0 \text{ if } P_s(\eta=1)=0 \text{ and } d_s=1} + \underbrace{\frac{\sum_{s=1}^{b} P_s(\eta = 1)\times d_s}{(a+b)}}_{=0 \text{ if } P_s(\eta=1)=1 \text{ and } d_s=0}. \end{aligned} \quad (8.4)$$

Lets first consider the first term on the right of Equation 8.4: As already described above, the provider cheats at a states, that is, he cheats at any states ($d_s = 1$) where he knows that he is

not tested ($P_s(\eta = 1) = 0$). Thus, the first term on the right of Equation 8.4 is zero, provided that the cloud provider has accurate knowledge about which states are *not* tested.

Now consider the second term on the right of Equation 8.4: The provider is compliant with certificate's controls at b states, i.e., he complies at any state ($d_s = 0$) where he knows that he is tested ($P_s(\eta = 1) = 1$). As a result, if the provider has complete knowledge about which states are tested, also the second term on the right of Equation 8.4 is zero.

At this point, we can summarize that our model OCP allows to show that the following intuition is true: If an opportunistic cloud provider has complete knowledge about which states are tested and the opportunistic limit of this provider is $\varphi = 0$, then the probability of detecting deceptive behavior of this provider is zero, that is, $D(\upsilon_{i,k}) = 0$.

The question at this point is how can we adopt our testing strategy in order to reduce the willingness for the cloud provider to cheat. A naive approach is to test the provider at every state. In this case, the provider knows that he is tested at every state and thus will not cheat. While theoretically a sound idea, it hardly provides any guidance for an approach applicable in practice because it implies that we can design tests which are capable of testing a cloud provider at every state when providing a cloud service. Also, Li et al. [129] points out that executing tests at every state is hard to apply in practice because it can incur significant overhead on the cloud service under test.

A feasible approach, however, to reduce the willingness of an opportunistic provider to deceive lies in randomization of tests. The key idea here is that the provider does not know whether he is tested at a certain state. More specifically, at any state the provider is either tested or not, that is,

$$\forall s \in \upsilon_{i,k} : P(\eta = 1) = 0.5.$$

We can now use Equation 8.4 to show the effect of randomly choosing at which states to test the provider. Let's start with considering the first term on the right of Equation 8.4: An opportunistic cloud provider cheating at least at one state means that $a > 0$. Since the provider can only guess with a probability of 0.5 that he is tested at a state at which he is deceiving, the first term on the right of Equation 8.4 is always greater than zero:

$$\frac{1}{(a+b)} \sum_{s=1}^{a} P_s(\eta = 1) \times d_s = \frac{1}{(a+b)} \sum_{s=1}^{a} 0.5 \times 1 > 0.$$

If the first term on the right of Equation 8.4 is always greater than zero, then probability of detecting deceptive behavior of the provider is always greater than zero, i.e., $D(\upsilon_{i,k}) > 0$.

Based on the above argument, we can draw the following conclusion: If the opportunistic limit of the provider is $\varphi = 0$, that is, he does not tolerate any probability of getting caught when cheating ($D(\upsilon_{i,k})$), then the provider will not deceive at all when states at which he is tested are chosen randomly. However, an opportunistic limit of $\varphi = 0$ is a special case and an opportunistic cloud provider may be willing to tolerate some probability of detection, that is, $\varphi > 0$. In these cases, the provider deceive at as many states a such that $D(\upsilon_{i,k}) \leq \varphi$.

We can conclude that based on our model OCP, randomization of tests reduces the number of states at which an opportunistic provider is willing to deceive. If the provider does not want to get caught cheating at all, then, with randomization of tests, he will not cheat at all.

8.3.2 Framework support of randomized tests

This section describes how our framework to design tests (see Section 4.3) supports randomization of tests on the level of test cases as well as on the level of test suites.

8.3.2.1 Test cases

Section 4.3.2 explained the role of *test cases* as one building bock of our framework to support continuous test-based cloud service certification. Among others, a test case is defined by its *input parameters* which, as laid out in Section 4.3.2, can be randomized.

As an example, consider a cloud provider supplying IaaS, i.e., virtual machines to customers having specified resources and performance, that is, providing minimum compute, memory, and storage capabilities. Amazon AWS, for example, offers specialized instances which are referred to as *compute optimized instances*[120] and which can be used for media transcoding, high-traffic web servers and other computationally-intensive applications.

Let's assume that some IaaS possesses a certificate which includes the control that *The promised minimum performance of virtual resources – usually defined as part of a SLA – has to always be satisfied*. In this context, consider a scenario where a customer's application uses the above IaaS to adapt to fluctuating daily workload by starting and stopping VMs as needed. Using our framework, we can design a continuous test which aims to check whether the allocated resource meet the minimum objective as specified in the SLA which, in turn, determines if the certificate's control is satisfied. To keep this example simple, we assume that test only consists of a single test case TC^{PT} takes as input parameter the list of newly started VMs, randomly selects one to conduct an application-specific performance test and compares the results with the expected performance which was previously established for the selected instance types. Furthermore, the test suite to which TC^{PT} is bound executes statically every 30 minutes.

In order to save costs, an opportunistic cloud provider may consider to deliberately *not* provide VMs in accordance with these SLA terms by, e.g., dedicating fewer than the minimum promised CPU resources of the host to a VM. Hence cost savings result from utilizing fewer physical resources required to satisfy the SLA terms and thus the certificate's control. The provider may use the remaining resources of a physical host to provide further IaaS to other customers. Continuing the above example, an opportunistic provider may consider to sometimes allocate fewer resource than promised to some instances which are temporarily added due to workload peaks of the customers' application.

Recall that the provider knows that he is subject to testing according to the certificate's performance control. Thus, if the provider does not want get caught when cheating on the promised resources for newly added VMs, he has to choose those VMs for which he assumes deceptive resource limitations are least likely to be detected. In case the cloud provider does not want to be caught cheating at all, that is, his opportunistic limit is $\varphi = 0$, then he will not cheat on the performance control at all. The reason for this is that – due to the randomization of the test case input parameter – he can only guess which of the newly started VM is selected for performance testing and thus cannot with certainty exclude the possibility that an instance is tested which does not comply with the performance control.

[120] `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html` [Accessed: 2018-12-13]

8.3.2.2 Test suites

In Section 4.3.3 of Chapter 4, we introduced *test suites* which are another building block of our framework to support continuous test-based certification of cloud services. One element of a test suite definition is the interval, that is, the period of time (in seconds) between two successive executions of test suites. As already pointed out in Section 4.3.3, this interval can either be static or serve as range from which the start of a test suite's execution is selected randomly.

Consider, as an example, that we use the continuous test CT^{LV} (see Section 5.3 of Chapter 5) to validate the location of a VM. An opportunistic provider may decide to migrate this VM to another location in order to save costs. However, the provider knows that he is subject to testing by CT^{LV}.

Let's assume that we seek to repeatedly validate the location of the VM at least once every six hours, that is, every $60 \times 60 \times 6 = 21600$ seconds. To that end, we can set the interval of the test suite TS_{probe} which collects new delay measurements on the Internet and Transport Layer to randomly select a value in the range of $[0, 21600]$ seconds. TS_{probe} will wait for this randomly selected time and then collect new probes.

If we assume that the delay measurement is trustworthy, that is, the cloud provider does not manipulate the measured delay, then VM will have to reside at our expected location at the time when the probes are collected. Otherwise the validation will fail, indicating that the provider does not satisfy a control of a guideline such as BSI C5 [31]. Therefore, if the provider does not want to get caught when deceitfully migrating and running the VM to an invalid location, then he has to choose a period in time during which he runs the VM at an invalid location where it least likely to get caught. Let's assume for the sake of our example that the provider does not want to get caught cheating at all, i.e., his opportunistic limit is $\varphi = 0$. In this case, since the execution of probes is randomly chosen within the six hours after the previous probing completed, the provider will refrain from running the VM at an invalid location at all but always host it at the expected location.

8.4 Summary and discussion

In this chapter, we presented a model of an adversarial cloud service provider who only pretends to comply with a set of controls. This *opportunistic cloud provider* only cheats, however, if he can be sure that he is not caught with a probability greater that his *opportunistic limit*. Our approach builds on the idea of the *explicit cheat formulation*, a variant of the *covert adversaries* which were introduced by Lindell and Aumann [44]. In order to be able to reason about the behavior of an opportunistic provider, we then combined the idea of covert adversaries with a modeling technique known as labeled transition systems (LTS).

Based on our model of an opportunistic provider's behavior, we then showed that randomization of tests can reduce the willingness of the provider to cheat. Furthermore, we explained how our framework to support continuous test-based cloud service certification introduced in Chapter 4 supports randomization of tests on the level of test cases as well as on the level of test suites.

One drawback of our approach is that it does not accurately reflect changing conditions over time. Assuming that in a real world setting, the majority of cases where a cloud service does not satisfy a certificate's controls, e.g., through outages and downtimes, occur rather unintended, that is, are the result of operational problems and are not intended, i.e., are

deliberately chosen by the provider. This means that the cloud service provider has to also factor in a certain percentage of non-compliance that will most likely occur over time, e.g., that there is a percentage of expected but unintended outages. On the one hand, this reduce the freedom of the cloud provider to intentionally not comply with a control. On the other hand, the provider will refrain from intentionally not complying at the beginning of an evaluation period because he does not know what events of unintended non-compliance may occur. Yet, towards the end of the interval, depending on the events of unintended non-compliance which actually did occur, he may choose to intentionally comply or not comply. Put differently: The choice of behavior of an opportunistic cloud provider is intrinsically time-sensitive. Extending our approach accordingly is subject to future work.

Another limitation of our approach is the strong assumption that a test is available which always correctly indicates whether a cloud provider complies with a control or just pretends to do so. Relaxing this assumption would require accounting for the cloud provider having cost-efficient means to deliberately manipulate test results, e.g., always respond to pings through a proxy instead of the actual cloud components endpoint, thereby hiding intentional outages. Thus, as part of future work, it is required to extend our approach with methods capable of establishing a trustworthy binding between a particular cloud service component under test and the measurement conducted by a test, for example, through remote attestation using Trusted Platform Modules (TPMs) as proposed by, e.g., Bertholon et al. [37].

Chapter 9

Conclusion

This chapter concludes this thesis by summarizing its contributions to the research challenges laid out in the introduction of this thesis (Section 9.1). Furthermore, we draw on the discussions provided at the end of Chapters 4, 5, 6, 7 and 8 and use identified drawbacks therein to point out directions of future research (Section 9.2).

9.1 Contributions to research challenges

In Section 1.2 of the introduction of this thesis, we described the main research challenges to overcome in order to develop methods to support continuous test-based cloud service certification. Hereafter, we revisit these challenges and explain how they are addressed by the contributions of this thesis.

Challenge 1: Design of tests supporting continuous cloud certification This challenge consists of developing a framework to design tests whose test results can serve as evidence to support continuous cloud service certification. Although there is no rigorous method available to bridge the *semantic gap*, i.e., rigorously derive evidence most suitable to check a specific control in an objective manner, using a framework to formulate tests as an *objective function* permits objective, repeatable and predictable production of evidence [40].

In this thesis, we addressed this challenge by introducing a framework which guides the design of tests, thereby supporting continuous cloud certification (Chapter 4). Our framework consists of the building blocks *test cases*, *test suites*, *workflow*, *test metrics* and *preconditions*; it allows designing tests to support validation of controls of cloud-specific as well as general standards and guidelines. It is independent of cloud service models and aims to flexibly integrate with existing cloud service infrastructures, preferably in a non-invasive or minimally invasive manner, that is, requiring only minimal to no changes to the infrastructure used to provide the cloud service under certification. The framework is built to reuse existing tools such as *Nmap*, *SQLMap*, or *Ping* in order to design tests. Also, existing approaches to test-based evidence production methods can leverage our framework to execute test repeatedly and reason about sequences of produced test results. Using our framework, *self-adaptive* tests can be designed which makes it possible to, e.g., react to changes of the environmental conditions within which a test operates and adapting it accordingly at runtime. Furthermore, our framework includes the proposition of four test metrics which are universally applicable to any continuous test supporting cloud service certification, independent of particular designs

of test cases, test suites or workflow. Also, we outlined one example implementation of our framework called *Clouditor*.

We demonstrated how our framework supports cloud service certification according to controls derived from BSI C5 [31], CSA STAR [23], and ISO/IEC 27001:2013 [24] based on five example test scenarios (Chapter 5). These example scenarios included testing the cloud service properties *availability*, *location*, *secure communication configuration*, *secure interface configuration*, and *user input validation*. However, a scenario-driven approach is limited in demonstrating the applicability of our framework in general. Therefore, we identified four general characteristics which we used to describe the test scenarios, thereby permitting us to draw conclusions about our framework's applicability that go beyond the scope of the example scenarios (Chapter 5).

Challenge 2: Definition of continuous tests This challenge lies in providing comparability of test results through the definition of continuous tests in a rigorous way. These definitions have to be *complete*, that is, contain all information required to configure tests while, at the same time, these definitions have to be agnostic to specific test implementations.

This thesis addressed this challenge through defining a domain specific language called *ConTest* (Chapter 6). This descriptive language allows to rigorously describe tests, providing a general representation of a test which is agnostic to a concrete implementation but abides to the building blocks of our framework presented in Chapter 4. Further, ConTest also serves a starting point to generate representations of test definitions which can be used to configure any specific test implementation. Thus, having a developer provide a code generator translating from ConTest to an implementation-specific configuration language ensures that the configuration of the test implementation follows the building blocks of our framework. We implemented ConTest and showed how it can be used to generate YAML configuration files which are used to configure tests within *Clouditor's engine*, one example implementation of our framework.

Challenge 3: Accuracy and precision of continuous test results This challenge consists of evaluating the accuracy and precision of continuous test results. This method has to allow us to compare alternative tests as well as to compare alternative test configurations.

In this thesis, we addressed this challenge by presenting a method to experimentally evaluate the accuracy and precision of continuous test results (Chapter 7). To that end, we built on the four universal test metrics introduced in Chapter 4 and defined what accuracy and precision mean in the context of continuous tests. Thereafter, we presented *control violations*, that is, sequences of events where each event manipulates a property of a cloud service under test such that the service does not fulfill one or more controls of a certificate. Thus, a control violation sequence establishes the ground truth which we then use to evaluate the accuracy and precision of a continuous test. Further, we explained how randomizing duration of as well as interval between control violation events allow for generalizing statements -— in the bounds of the range from which random values are selected — about a continuous test's accuracy and precision.

At the heart of our method are *accuracy* and *precision measures* which are based on the universal test metrics. We derived these measures by combining the notion of control violation sequences with our definition of continuous test's accuracy based on the universal test metrics. These measures allow us to make statements about, e.g., the average error a particular test makes when estimating the duration of a control violation event. Lastly, in

order to demonstrate the application of our method, we presented three example scenarios to show how the accuracy and precision measures allow us to compare alternative tests as well as alternative test configurations to select the one most suited.

Challenge 4: Trustworthy continuous test results If we use tests to produce evidence supporting continuous cloud certification, then not trusting the cloud service provider whose service is under test implies that we cannot unconditionally trust results of tests. Thus, this challenge lies in providing an approach how to adapt tests to produce trustworthy test results when faced with a fraudulent provider.

This thesis addressed this challenge by presenting a model of an adversarial cloud service provider who only pretends to comply with a set of controls (Chapter 8). This *opportunistic cloud provider* only cheats, however, if he can be sure that he is not caught with a certain probability above his so-called *opportunistic limit*. Our approach builds on the idea of the *explicit cheat formulation*, a variant of the *covert adversaries* which were introduced by Lindell and Aumann [44]. In order to be able to reason about the behavior of an opportunistic provider, we then combined the idea of covert adversaries with a modeling technique known as *Labeled Transition Systems (LTS)*.

Based on our model of an opportunistic provider's behavior, we were able to show that randomization of tests can reduce the willingness of the opportunistic provider to cheat. Also, we explained how our framework to support continuous test-based cloud service certification introduced in Chapter 4 supports randomization of tests on the level of test cases as well as on the level of test suites.

9.2 Directions for future research

In this section, we point out directions of future research in the field of continuous test-based cloud service certification.

A security model for cloud certification systems Continuous tests seek to support continuous cloud certification which, in turn, aims at increasing customer's trust in and transparency of cloud services. However, if not properly secured, such mechanisms can also leak critical information which can be used by adversaries. As Santos et al. [246] point out, information primarily produced to increase transparency can be also used to trace vulnerabilities of a cloud infrastructure. As an example, consider the test scenario to continuously check *user input validation* which we introduced as part of Chapter 5 to demonstrate applicability of our framework. In this scenario, we check whether a SaaS application is vulnerable to SQL Injections (SQLI). It is obvious that leaking test results indicating a SQLI vulnerability can simplify the steps an attacker has to take to successfully attack the service. Therefore, it is vital to secure the system which implements continuous tests of cloud services. The underlying research question here is:

> *How to assess and mitigate the risk of using continuous test-based certification to validate that a cloud service complies with a set of controls?*

This risk analysis has to guide derivation of a suitable security model, that is, a set of security measures to be integrated into the certification system's architecture which aim at preventing or mitigating identified risks. Since our framework to design tests does not mandate a specific

architecture, it does not impose any restrictions on a certification system's security model. At the same time, the building blocks of our framework serves as a starting point to identify assets worth protecting, e.g., produced evidence in form of measurements computed by test metrics.

Risk-based integration of continuous tests One important requirement of our framework to design tests presented in Chapter 4 is that tests should flexibly integrate with existing cloud infrastructures. Here, we brought forward the notion of *minimally invasive* testing, that is, continuous tests which only require minimal to no changes to the infrastructure of the cloud service under test. Although our framework also supports invasive testing, we followed this notion of minimally invasive testing throughout all of our example test scenarios presented in Chapter 5.

However, only considering minimally invasive test designs has limitations as to which evidence can be collected. As a result, the quality of statements and possibly also the scope of controls which can be evaluated using minimally invasive continuous tests is limited as well. Yet designing invasive continuous tests which, e.g., require deploying and operate additional components within the infrastructure of the cloud service also increase the probability of vulnerabilities introduced by these structural changes. Therefore, as part of future research, an approach is needed to describe the restrictions inherent to minimally invasive testing and investigate how removing these restrictions through invasive testing increases the overall risk exposure of the cloud service. The underlying research question here is:

> *What extent of invasive continuous testing is required to check a maximum number of controls while retaining a predefined threshold of risk exposure of the cloud service under certification?*

This approach would be orthogonal to the work of this thesis since our framework can also be used to design different degrees of invasive tests as needed.

Conformance clauses for cloud service certificates Recall that as part of *Research Challenge 1: Design of tests to support continuous cloud certification*, we highlighted that there is no method available to objectively derive evidence best suited to check a specific control. What is suitable evidence to check a control is thus left to consensus among domain experts which Maibaum and Wassyng [40] refer to as an *agreed-upon objective function* to measure a certain property of a software under certification. Yet, as of today, no *standard set* of agreed-upon objective functions is available which can be used to check if a cloud service satisfies a certificate's controls. The decisions whether suitable evidence exists and if this evidence indicates that a control is satisfied is left to the possibly well-informed but subjective opinion of a human auditor. However, *subjectivity* per se is not the problem, the problem is that the auditors decisions is *implicit*, that is, not formulated in an objectively conceivable manner, thereby preventing repeated and predictable evidence production and interpretation. Thus, the underlying research question here is:

> *How can we formulate production and interpretation of evidence to standardize the evaluation of a specific control in an objective, repeatable and predictable manner?*

A starting point to answer this question are *conformance clauses* which we described in Section 2.2.3 of the background chapter of this thesis. These clauses define how to test

conformance of a software product with a standard or specification. This includes to define, on a technical level, how to test conformance, including, e.g., what tools to use. Our domain-specific language *ConTest* presented in Chapter 6 can support the construction of conformance clause templates where conformance of a cloud service is evaluated using continuous tests.

Analysis of side-effects of continuous tests As already pointed out in the introduction as well as in the background chapter of this thesis, continuously testing whether a cloud service conforms with a set of controls is not to be understood in a strict mathematical sense: No matter how sophisticated the method to produce evidence, producing evidence will always be — in a strict mathematical sense — discrete tasks that occur at some point in time.

In Chapter 7 we showed that, in some cases, the accuracy of continuous tests results increases if the interval between repeated tests decreases. Naturally, this is expected since if the frequency of executed tests increases, so does the probability to detect a property violation leading to non-compliance with a certificate's control. Yet accuracy and precision of a continuous test may not be the only two characteristics needed to select the most suitable continuous test in a real world scenario. Another important property of a test is the overhead which it imposes on the service under test, especially when facing multiple, possibly concurrent tests. As pointed out by Li et al. [129], uninterruptedly testing cloud service providers can incur intolerable overhead on the cloud service under test. An interesting idea here is to measure the overhead caused by tests and then select those tests and configurations which incur minimal overhead while retaining required accuracy and precision of results.

Aside from performance overhead incurred through continuous tests, there are other side-effects which need to be considered. For example, security tools such as *SQLMap* actually exploit a detected SQLI vulnerability which may lead to exposure of sensitive information, e.g., personal data. Further, consider continuously testing the available bandwidth of a cloud service component which may significantly increase operational costs. As a last example, continuous tests aiming to evaluate the robustness of a cloud service may unintentionally disrupt regular service operation, thereby creating a financial loss through service downtime. In summary, this leads to the following research question which is subject to future work:

> *What side-effects can continuous tests cause and how can these side-effects be mitigated to not exceed a predefined threshold of risk exposure of a cloud service under test?*

Bibliography

[1] M. Utting and B. Legeard, *Practical model-based testing: A tools approach*. Morgan Kaufmann, 2010.

[2] Michael Felderer, Matthias Büchlein, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner, “Security Testing: A Survey,” *Advances in Computers*, vol. 101, pp. 1–51, 2016.

[3] “AWS Global Infrastructure.” [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/?nc1=h_ls, [Accessed: 2018-12-13].

[4] Stanford Encyclopedia of Philosophy, “The Modern History of Computing.” [Online]. Available: https://plato.stanford.edu/entries/computing-history/#Bab, June 2006. [Accessed: 2018-12-13].

[5] C. Babbage, *On the economy of machinery and manufactures*. Cambridge University Press, 1832.

[6] A. Giddens and D. Held, *Classes, Power, and Conflict: Classical and Contemporary debates*. University of California Press, 1982.

[7] U. Arnold, “New dimensions of outsourcing: A combination of transaction cost economics and the core competencies concept,” *European Journal of Purchasing & Supply Management*, vol. 6, no. 1, pp. 23–29, 2000.

[8] M. C. Lacity, L. P. Willcocks, and D. F. Feeny, “The value of selective IT sourcing,” *Sloan management review*, vol. 37, no. 3, p. 13, 1996.

[9] H. R. Motahari Nezhad, B. Stephenson, and S. Singhal, “Outsourcing business to cloud computing services: Opportunities and challenges,” *IEEE Internet Computing*, vol. 10, no. 4, pp. 1–17, 2009.

[10] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” *NIST Special Publication*, vol. 800, no. 145, p. 7, 2011.

[11] European Commission, “Unleashing the Potential of Cloud Computing in Europe.” [Online]. Available: http://europa.eu/rapid/press-release_MEMO-12-713_en.htm, September 2012. [Accessed: 2018-12-13].

[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, “A view of cloud computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[13] D. Catteddu and G. Hogben, "Cloud computing risk assessment," *European Network and Information Security Agency (ENISA)*, pp. 583–592, 2009.

[14] M. Stieninger and D. Nedbal, "Diffusion and acceptance of cloud computing in SMEs: Towards a valence model of relevant factors," in *47th Hawaii International Conference on System Sciences (HICSS)*, pp. 3307–3316, IEEE, 2014.

[15] M. Stieninger, D. Nedbal, W. Wetzlinger, G. Wagner, and M. A. Erskine, "Impacts on the organizational adoption of cloud computing: A reconceptualization of influencing factors," *Procedia Technology*, vol. 16, pp. 85–93, 2014.

[16] A. Bisong, M. Rahman, *et al.*, "An overview of the security concerns in enterprise cloud computing," *arXiv preprint arXiv:1101.5613*, 2011.

[17] S. Pearson, "Toward accountability in the cloud," *IEEE Internet Computing*, vol. 15, no. 4, pp. 64–69, 2011.

[18] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of network and computer applications*, vol. 34, no. 1, pp. 1–11, 2011.

[19] A. Sunyaev and S. Schneider, "Cloud services certification," *Communications of the ACM*, vol. 56, no. 2, pp. 33–36, 2013.

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

[21] International Organization for Standardization (ISO), "ISO/IEC 27000:2018 Information technology – Security techniques – Information security management systems – overview and vocabulary." Available: `https://www.iso.org/standard/73906.html`, Feb. 2018. [Accessed: 2018-12-13].

[22] Cloud Security Alliance (CSA), "Cloud Control Matrix (CCM, Version 3.0.1): Security Controls Framework for Cloud Providers & Consumers." Available: `https://cloudsecurityalliance.org/research/ccm/`, Sept. 2017. [Accessed: 2018-12-13].

[23] Cloud Security Alliance (CSA), "Security, Trust and Assurance Registry (STAR)." [Online]. Available: `https://cloudsecurityalliance.org/star/certification/`. [Accessed: 2018-12-13].

[24] International Organization for Standardization (ISO), "ISO/IEC 27001:2013 Information technology – Security techniques – Information security management systems – Requirements." Available: `https://www.iso.org/standard/54534.html`, Oct. 2013. [Accessed: 2018-12-13].

[25] International Organization for Standardization (ISO), "ISO 9000:2015 Quality management systems – Fundamentals and vocabulary." Available: `https://www.iso.org/standard/45481.html`, Sept. 2015. [Accessed: 2018-12-13].

[26] I. Windhorst and A. Sunyaev, "Dynamic certification of cloud services," in *8th International Conference on Availability, Reliability and Security (ARES)*, pp. 412–417, IEEE, 2013.

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

[28] M. Krotsiani, G. Spanoudakis, and K. Mahbub, "Incremental certification of cloud services," in *7th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*, pp. 72–80, 2013.

[29] S. Lins, S. Schneider, and A. Sunyaev, "Trust is Good, Control is Better: Creating Secure Clouds by Continuous Auditing," in *IEEE Transactions on Cloud Computing*, 2016.

[30] "The Federal Risk and Authorization Management Program (FedRAMP)." [Online]. Available: https://www.fedramp.gov/. [Accessed: 2018-12-13].

[31] The German Federal Office for Information Security (Bundesamt für Informationssicherheit (BSI)), "Cloud Computing Compliance Controls Catalogue (C5)." Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/CloudComputing/ComplianceControlsCatalogue-Cloud_Computing-C5.pdf?__blob=publicationFile&v=3, 2016. [Accessed: 2018-12-13].

[32] National Institute of Standards and Technology (NIST), "Special Publication 800-37." Available: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-37r1.pdf, Feb. 2010. [Accessed: 2018-12-13].

[33] M. Krotsiani and G. Spanoudakis, "Continuous Certification of Non-repudiation in Cloud Storage Services," in *13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 921–928, IEEE, 2014.

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

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

[36] Trusted Computing Group (TCG), "Trusted Platform Module (TPM) Summary." [Online]. Available: https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/. [Accessed: 2018-12-13].

[37] B. Bertholon, S. Varrette, and P. Bouvry, "Certicloud: A novel tpm-based approach to ensure cloud IaaS security," in *International Conference on Cloud Computing (CLOUD)*, pp. 121–130, IEEE, 2011.

[38] A. Muñoz and A. Mana, "Bridging the gap between software certification and trusted computing for securing cloud computing," in *9th World Congress on Services (SERVICES)*, pp. 103–110, IEEE, 2013.

[39] S. Schneider, J. Lansing, F. Gao, and A. Sunyaev, "A Taxonomic Perspective on Certification Schemes: Development of a Taxonomy for Cloud Service Certification Criteria," in *47th Hawaii International Conference on System Sciences (HICSS)*, pp. 4998–5007, IEEE, 2014.

[40] T. Maibaum and A. Wassyng, "A product-focused approach to software certification," *Computer*, vol. 41, no. 2, 2008.

[41] J. Lansing, S. Schneider, and A. Sunyaev, "Cloud Service Certifications: Measuring Consumers' Preferences For Assurances," in *European Conference on Information Systems (ECIS)*, p. 181, 2013.

[42] B. Sturm, J. Lansing, and A. Sunyaev, "Moving in the right direction? Mapping literature on cloud service certifications' outcomes with practitioners' perceptions," *22nd European Conference on Information Systems (ECIS)*, 2014.

[43] S. Lins, "The Effect of Continuous Cloud Service Certification on Cloud Service Customers," in *Proceedings of 13. Internationale Tagung Wirtschaftsinformatik (WI)*, 2017.

[44] Y. Aumann and Y. Lindell, "Security against covert adversaries: Efficient protocols for realistic adversaries," *Journal of Cryptology*, vol. 23, no. 2, pp. 281–343, 2010.

[45] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop (GCE)*, pp. 1–10, IEEE, 2008.

[46] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation computer systems*, vol. 25, no. 6, pp. 599–616, 2009.

[47] J. Geelan *et al.*, "Twenty one experts define cloud computing," *Cloud Computing Journal*, vol. 4, pp. 1–5, 2009.

[48] E. Knorr and G. Gruman, "What cloud computing really means," *InfoWorld*, vol. 7, pp. 20–20, 2008.

[49] E. Hand, "Head in the clouds: Cloud computing is being pitched as a new nirvana for scientists drowning in data. But can it deliver?," *Nature*, vol. 449, no. 7165, pp. 963–964, 2007.

[50] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing," in *Grid Computing Environments Workshop (GCE)*, pp. 1–10, IEEE, 2008.

[51] R. B. Bohn, J. Messina, F. Liu, J. Tong, and J. Mao, "NIST cloud computing reference architecture," in *World Congress on Services (SERVICES)*, pp. 594–596, IEEE, 2011.

[52] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf, "NIST cloud computing reference architecture," *NIST special publication*, vol. 500, p. 292, 2011.

[53] Cloud Security Alliance (CSA), "Enterprise Architecture Working Group." [Online]. Available: `https://research.cloudsecurityalliance.org/tci/`. [Accessed: 2018-12-13].

[54] R. Stifani, S. Pappe, G. Breiter, and M. Behrendt, "IBM Cloud Computing Reference Architecture," *IBM Academy of Technology, Academy TechNotes (ATN)*, vol. 3, no. 1, 2012.

[55] L. M. Vaquero, L. Rodero Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2008.

[56] Cloud Security Alliance (CSA), "Custom Applications and IaaS Trends 2017." Available: `https://downloads.cloudsecurityalliance.org/assets/survey/custom-applications-and-iaas-trends-2017.pdf`, 2017. [Accessed: 2018-12-13].

[57] "Standard Glossary of Terms used in Software Testing – Version 3.1," *International Software Testing Qualifications Board (ISTQB)*, 2010.

[58] P. Bourque, R. E. Fairley, *et al.*, *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE, 2014.

[59] E. Damiani, C. A. Ardagna, and N. El Ioini, *Open source systems security certification*. Springer Science & Business Media, 2008.

[60] International Organization for Standardization (ISO), "ISO/IEC/IEEE 24765:2017 – Systems and Software Engineering–Vocabulary." Available: `https://www.iso.org/standard/71952.html`, Sept. 2017. [Accessed: 2018-12-13].

[61] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[62] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, 1972.

[63] Martin Fowler, "ComponentTest." [Online]. Available: `https://martinfowler.com/bliki/ComponentTest.html`, April 2013. [Accessed: 2018-12-13].

[64] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[65] Martin Fowler, "TestCoverage." [Online]. Available: `https://martinfowler.com/bliki/TestCoverage.html`, April 2012. [Accessed: 2018-12-13].

[66] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *14th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 281–292, IEEE, 2003.

[67] D. Saff and M. D. Ernst, “An experimental evaluation of continuous testing during development,” in *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 76–85, ACM, 2004.

[68] D. Saff and M. D. Ernst, “Continuous testing in Eclipse,” in *Proceedings of the 27th International Conference on Software Engineering*, pp. 668–669, ACM, 2005.

[69] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[70] B. Potter and G. McGraw, “Software security testing,” *IEEE Security & Privacy*, vol. 2, no. 5, pp. 81–85, 2004.

[71] I. Alexander, “Misuse cases: Use cases with hostile intent,” *IEEE software*, vol. 20, no. 1, pp. 58–66, 2003.

[72] P. Zech, M. Felderer, M. Farwick, and R. Breu, “A Concept for Language-Oriented Security Testing,” in *7th International Conference on Software Security and Reliability-Companion (SERE-C)*, pp. 53–62, IEEE, 2013.

[73] G. Tian yang, S. Yin Sheng, and F. You yuan, “Research on software security testing,” *World Academy of Science, Engineering and Technology*, vol. 70, pp. 647–651, 2010.

[74] H. Herbert *et al.*, “Why security testing is hard,” *IEEE Security & Privacy*, 2003.

[75] M. Howard and D. LeBlanc, *Writing secure code*. Pearson Education, 2003.

[76] V. B. Livshits and M. S. Lam, “Finding Security Vulnerabilities in Java Applications with Static Analysis,” in *Usenix Security*, vol. 14, 2005.

[77] P. Louridas, “Static code analysis,” *IEEE Software*, vol. 23, no. 4, pp. 58–61, 2006.

[78] C. Cowan, M. Barringer, S. Beattie, G. Kroah Hartman, M. Frantzen, and J. Lokier, “FormatGuard: Automatic Protection From printf Format String Vulnerabilities,” in *USENIX Security Symposium*, vol. 91, 2001.

[79] M. D. Ernst, “Static and dynamic analysis: Synergy and duality,” in *ICSE Workshop on Dynamic Analysis (WODA)*, pp. 24–27, 2003.

[80] W. G. Halfond and A. Orso, “AMNESIA: Analysis and monitoring for NEutralizing SQL-injection attacks,” in *Proceedings of the 20th International Conference on Automated Software Engineering*, pp. 174–183, ACM, 2005.

[81] K. Henry, *Penetration Testing: Protecting Networks and Systems*. IT Governance Publishing, 2012.

[82] B. Arkin, S. Stender, and G. McGraw, “Software penetration testing,” *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84–87, 2005.

[83] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[84] C. Miller and Z. N. Peterson, “Analysis of mutation and generation-based fuzzing,” *Independent Security Evaluators, Tech. Rep*, 2007.

[85] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated Whitebox Fuzz Testing," in *NDSS*, vol. 8, pp. 151–166, 2008.

[86] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[87] J. Newsome and D. Song, "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *Proceedings of the 12th Network and Distributed Systems Security Symposium*, 2005.

[88] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 135–148, IEEE, 2006.

[89] W. G. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 175–185, ACM, 2006.

[90] National Institute of Standards and Technology (NIST), "Overview of Conformance Testing." [Online]. Available: `https://www.nist.gov/itl/ssd/information-systems-group/overview-conformance-testing`, January 1999. [Accessed: 2018-12-13].

[91] International Organization for Standardization (ISO), "ISO/IEC 17000: 2004 Conformity Assessment–Vocabulary and General Principles." Available: `https://www.iso.org/standard/29316.html`, Nov. 2004. [Accessed: 2018-12-13].

[92] M. M. Eloff and S. H. von Solms, "Information security management: A hierarchical framework for various approaches," *Computers & Security*, vol. 19, no. 3, pp. 243–256, 2000.

[93] IATAC and DACS, *Software Security Assurance: State of the Art Report (SOAR)*, July 2007. Available: `http://www.dtic.mil/docs/citations/ADA472363`, [Accessed: 2018-12-13].

[94] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, "From Security to Assurance in the Cloud: A Survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 2, 2015.

[95] National Institute of Standards and Technology (NIST), "Special Publication 800-53." Available: `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf`, Apr. 2013. [Accessed: 2018-12-13].

[96] International Organization for Standardization (ISO), "ISO 9001:2015 Quality management systems – Requirements." Available: `https://www.iso.org/standard/62085.html`, Sept. 2015. [Accessed: 2018-12-13].

[97] Software Engineering Institute (SEI), "Capability Maturity Model Integration (CMMI)." [Online]. Available: `https://www.sei.cmu.edu/cmmi/`. [Accessed: 2018-12-13].

[98] H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, no. 5, pp. 19–25, 1987.

[99] J. Voas, "The software quality certification triangle," *Crosstalk*, vol. 11, no. 11, pp. 12–14, 1998.

[100] J. Voas, "Developing a usage-based software certification process," *Computer*, vol. 33, no. 8, pp. 32–37, 2000.

[101] J. Morris, G. Lee, K. Parker, G. A. Bundell, and C. P. Lam, "Software component certification," *Computer*, vol. 34, no. 9, pp. 30–36, 2001.

[102] "Common Criteria for Information Technology Security Evaluation (CC)." [Online]. Available: `https://www.commoncriteriaportal.org/cc/`. [Accessed: 2018-12-13].

[103] D. S. Herrmann, *Using the Common Criteria for IT security evaluation*. CRC Press, 2002.

[104] A. Bicego and P. Kuvaja, "Software process maturity and certification," *Journal of Systems Architecture*, vol. 42, no. 8, pp. 611–620, 1996.

[105] International Organization for Standardization (ISO), "ISO/IEC 33001:2015 Information technology – Process assessment – Concepts and terminology." Available: `https://www.iso.org/standard/54175.html`, Mar. 2015. [Accessed: 2018-12-13].

[106] T. Erl, *SOA: Principles of service design*. Prentice Hall Press, 2007.

[107] M. P. Papazoglou, V. Andrikopoulos, and S. Benbernou, "Managing evolving services," *IEEE Software*, vol. 28, no. 3, pp. 49–55, 2011.

[108] M. Anisetti, C. A. Ardagna, E. Damiani, and F. Saonara, "A test-based security certification scheme for web services," *ACM Transactions on the Web (TWEB)*, vol. 7, no. 2, p. 5, 2013.

[109] M. Anisetti, C. Ardagna, and E. Damiani, "Toward certification of services," in *International Workshop on Business System Management and Engineering (BSME)*, 2010.

[110] OASIS Standard, "Web Services Security v1.1.1." [Online]. Available: `https://www.oasis-open.org/standards#wssv1.1.1`, May 2012. [Accessed: 2018-12-13].

[111] OASIS Standard, "WS-Reliability 1.1." [Online]. Available: `https://www.oasis-open.org/standards#wsrv1.1`, Nov. 2004. [Accessed: 2018-12-13].

[112] M. Anisetti, C. A. Ardagna, E. Damiani, F. Frati, H. A. Müller, and A. Pahlevan, "Web service assurance: The notion and the issues," *Future Internet*, vol. 4, no. 1, pp. 92–109, 2012.

[113] M. Anisetti, C. A. Ardagna, F. Guida, S. Gürgens, V. Lotz, A. Maña, C. Pandolfo, J.-C. Pazzaglia, G. Pujol, and G. Spanoudakis, "ASSERT4SOA: Toward security certification of service-oriented applications," in *On the Move to Meaningful Internet Systems (OTM)*, pp. 38–40, Springer, 2010.

[114] EuroCloud Europe (ECE), "EuroCloud Star Audit (ECSA)." [Online]. Available: https://staraudit.org/. [Accessed: 2018-12-13].

[115] D. Catteddu, G. Hogben, *et al.*, "Cloud computing information assurance framework," *European Network and Information Security Agency (ENISA)*, 2009. Available: https://www.enisa.europa.eu/publications/cloud-computing-information-assurance-framework/at_download/fullReport, [Accessed: 2018-12-13].

[116] R. Bernnat, W. Zink, N. Bieber, S. Tai, J. Strach, and R. Fischer, "The Standardisation Environment for Cloud Computing." Available: https://www.trusted-cloud.de/sites/default/files/trustedcloud-studie-standardisation-environment-for-cloud-computing.pdf, Feb. 2012. [Accessed: 2018-12-13].

[117] National Institute of Standards and Technology (NIST), "SP 500-291, Version 2: Cloud Computing Standards Roadmap." Available: https://www.nist.gov/sites/default/files/documents/itl/cloud/NIST_SP-500-291_Version-2_2013_June18_FINAL.pdf, July 2013. [Accessed: 2018-12-13].

[118] K. M. Khan and Q. Malluhi, "Establishing trust in cloud computing," *IT professional*, vol. 12, no. 5, pp. 20–27, 2010.

[119] R. K. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee, "TrustCloud: A framework for accountability and trust in cloud computing," in *7th World Congress on Services (SERVICES)*, pp. 584–588, IEEE, 2011.

[120] P. Stephanow, C. Banse, and J. Schütte, "Generating Threat Profiles for Cloud Service Certification Systems," in *17th High Assurance Systems Engineering Symposium (HASE)*, pp. 260–267, IEEE, 2016.

[121] E. Damiani, G. Spanoudakis, S. Katopodis, K. Mahbub, M. Krotsiani, S. Cimato, M. Anisetti, C. Ardagna, F. Zavatarelli, M. Rosa, V. Alvarez, R. Menicocci, A. Riccardi, V. Bagini, J. Espinar, A. Muñoz, A. Maña, and H. Koshutanski, "CUMULUS Deliverable 2.4: Certification models." Available: https://cordis.europa.eu/docs/projects/cnect/0/318580/080/deliverables/001-D24FinalCUMULUSCertificationModelsv2.pdf, Dec. 2015. [Accessed: 2018-12-13].

[122] Common Criteria (CC), "Assurance Continuity: CCRA Requirements (Version 2.1)." Available: https://www.commoncriteriaportal.org/files/operatingprocedures/2012-06-01.pdf, June 2012. [Accessed: 2018-12-13].

[123] R. Harjani, A. Maña, M. Arjona, A. Muñoz, J. Espinar, and H. Koshutanski, "An Integrated Framework for Multi-layer Certification-based Assurance," in *8th Layered Assurance Workshop (LAW)*, 2014.

[124] P. Stephanow and M. Gall, "Language Classes for Cloud Service Certification Systems," in *11th World Congress on Services (SERVICES)*, pp. 127–134, IEEE, 2015.

[125] P. Stephanow and N. Fallenbeck, "Towards continuous certification of Infrastructure-as-a-Service using low-level metrics," in *12th International Conference on Ubiquitous Intelligence and Computing, 12th International Conference on Autonomic and Trusted Computing and 15th International Conference on Scalable Computing and Communications and its associated Workshops (UIC-ATC-ScalCom)*, pp. 1485–1492, IEEE, 2015.

[126] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger, "Cloud verifier: Verifiable auditing service for iaas clouds," in *9th World Congress on Services (SERVICES)*, pp. 239–246, IEEE, 2013.

[127] P. Stephanow and C. Banse, "Evaluating the performance of continuous test-based cloud service certification," in *17th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 1117–1126, IEEE/ACM, 2017.

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

[129] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing public cloud providers," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, pp. 1–14, ACM, 2010.

[130] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, pp. 143–154, ACM, 2010.

[131] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *Proceedings of the 1st USENIX conference on Hot topics in cloud computing (HotCloud)*, vol. 9, pp. 3–3, 2009.

[132] Cloud Security Alliance (CSA), "Consensus Assessments Initiative Questionnaire (CAIQ, Version 3.0.1)." [Online]. Available: `https://cloudsecurityalliance.org/download/consensus-assessments-initiative-questionnaire-v3-0-1/`, Sept. 2017. [Accessed: 2018-12-13].

[133] EuroCloud Europe, "StarAudit Assessment Tool." Available: `https://staraudit.org/assessment/`, [Accessed: 2018-12-13].

[134] S. Katopodis, G. Spanoudakis, and K. Mahbub, "Towards hybrid cloud service certification models," in *International Conference on Services Computing (SCC)*, pp. 394–399, IEEE, 2014.

[135] M. Anisetti, C. Ardagna, E. Damiani, and F. Gaudenzi, "A semi-automatic and trustworthy scheme for continuous cloud service certification," *IEEE Transactions on Services Computing*, 2017.

[136] M. Anisetti, C. Ardagna, E. Damiani, *et al.*, "A Certification-Based Trust Model for Autonomic Cloud Computing Systems," in *International Conference on Cloud and Autonomic Computing (ICCAC)*, pp. 212–219, IEEE, 2014.

[137] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

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

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

[140] D. Gonzales, J. Kaplan, E. Saltzman, Z. Winkelman, and D. Woods, "Cloud-trust–a security assessment model for infrastructure as a service (IaaS) clouds," *IEEE Transactions on Cloud Computing*, 2015.

[141] S. Bleikertz, M. Schunter, C. W. Probst, D. Pendarakis, and K. Eriksson, "Security audits of multi-tier virtual infrastructures in public infrastructure clouds," in *Proceedings of the ACM workshop on Cloud computing security*, pp. 93–102, ACM, 2010.

[142] M. Whaiduzzaman and A. Gani, "Measuring security for cloud service provider: A Third Party approach," in *International Conference on Electrical Information and Communication Technology (EICT)*, pp. 1–6, IEEE, 2014.

[143] P. Zech, "Risk-based security testing in cloud computing environments," in *4th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 411–414, IEEE, 2011.

[144] P. Zech, M. Felderer, and R. Breu, "Towards a model based security testing approach of cloud computing environments," in *6th International Conference on Software Security and Reliability Companion (SERE-C)*, pp. 47–56, IEEE, 2012.

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

[146] M. Eskandari, A. S. De Oliveira, and B. Crispo, "VLOC: An Approach to Verify the Physical Location of a Virtual Machine In Cloud," in *6th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 86–94, IEEE, 2014.

[147] C. Jaiswal and V. Kumar, "IGOD: Identification of geolocation of cloud datacenters," *Journal of Information Security and Applications*, vol. 27, pp. 85–102, 2016.

[148] M. Fotouhi, A. Anand, and R. Hasan, "PLAG: Practical Landmark Allocation for Cloud Geolocation," in *8th International Conference on Cloud Computing (CLOUD)*, pp. 1103–1106, IEEE, 2015.

[149] B. Gueye, A. Ziviani, M. Crovella, and S. Fdida, "Constraint-based geolocation of internet hosts," *IEEE/ACM Transactions On Networking*, vol. 14, no. 6, pp. 1219–1232, 2006.

[150] Z. N. Peterson, M. Gondree, and R. Beverly, "A Position Paper on Data Sovereignty: The Importance of Geolocating Data in the Cloud," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[151] M. Gondree and Z. N. Peterson, "Geolocation of data in the cloud," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, pp. 25–36, ACM, 2013.

[152] D. L. Fu, X. G. Peng, and Y. L. Yang, "Trusted validation for geolocation of cloud data," *The Computer Journal*, vol. 58, no. 10, pp. 2595–2607, 2015.

[153] A. Albeshri, C. Boyd, and J. G. Nieto, "Geoproof: Proofs of geographic location for cloud computing environment," in *32nd International Conference on Distributed Computing Systems Workshops*, pp. 506–514, IEEE, 2012.

[154] T. Ries, V. Fusenig, C. Vilbois, and T. Engel, "Verification of data location in cloud networking," in *4th International Conference on Utility and Cloud Computing (UCC)*, pp. 439–444, IEEE, 2011.

[155] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," in *ACM SIGCOMM Computer Communication Review*, vol. 34, pp. 15–26, ACM, 2004.

[156] Y. Chen, Y. Xiong, X. Shi, B. Deng, and X. Li, "Pharos: A decentralized and hierarchical network coordinate system for internet distance prediction," in *Global Telecommunications Conference (GLOBECOM)*, pp. 421–426, IEEE, 2007.

[157] Y. Chen, X. Wang, X. Song, E. K. Lua, C. Shi, X. Zhao, B. Deng, and X. Li, "Phoenix: Towards an accurate, practical and decentralized network coordinate system," in *International Conference on Research in Networking*, pp. 313–325, Springer, 2009.

[158] V. N. Padmanabhan and L. Subramanian, "An investigation of geographic mapping techniques for Internet hosts," in *ACM SIGCOMM Computer Communication Review*, vol. 31, pp. 173–185, ACM, 2001.

[159] E. Katz Bassett, J. P. John, A. Krishnamurthy, D. Wetherall, T. Anderson, and Y. Chawathe, "Towards IP geolocation using delay and topology measurements," in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pp. 71–84, ACM, 2006.

[160] B. Eriksson, P. Barford, J. Sommers, and R. Nowak, "A learning-based approach for IP geolocation," in *International Conference on Passive and Active Network Measurement*, pp. 171–180, Springer, 2010.

[161] M. J. Arif, S. Karunasekera, and S. Kulkarni, "GeoWeight: Internet host geolocation based on a probability model for latency measurements," in *Proceedings of the 33rd Australasian Conferenc on Computer Science*, vol. 102, pp. 89–98, Australian Computer Society, 2010.

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

[163] R. Houlihan, X. Du, C. C. Tan, J. Wu, and M. Guizani, “Auditing cloud service level agreement on VM CPU speed,” in *International Conference on Communications (ICC)*, pp. 799–803, IEEE, 2014.

[164] L. Ye, H. Zhang, J. Shi, and X. Du, “Verifying cloud service level agreement,” in *Global Communications Conference (GLOBECOM)*, pp. 777–782, IEEE, 2012.

[165] F. Koeppe and J. Schneider, “Do you get what you pay for? Using proof-of-work functions to verify performance assertions in the cloud,” in *2nd International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 687–692, IEEE, 2010.

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

[167] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, “D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology,” in *10th International Conference on Cluster, Cloud and Grid Computing (CCGrid), 2010*, pp. 631–636, IEEE, 2010.

[168] C. Pham, D. Chen, Z. Kalbarczyk, and R. K. Iyer, “Cloudval: A framework for validation of virtualization environment in cloud infrastructure,” in *41st International Conference on Dependable Systems & Networks (DSN)*, pp. 189–196, IEEE, 2011.

[169] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, “NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors,” in *Proceedings of the International Computer Performance and Dependability Symposium (IPDS)*, pp. 91–100, IEEE, 2000.

[170] M. Le, A. Gallagher, and Y. Tamir, “Challenges and opportunities with fault injection in virtualized systems,” in *1st International Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, 2008.

[171] W. Lu, X. Hu, S. Wang, and X. Li, “A multi-criteria QoS-aware trust service composition algorithm in cloud computing environments,” *International Journal of Grid and Distributed Computing*, vol. 7, no. 1, pp. 77–78, 2014.

[172] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Q. Wang, J. Park, and C. Pu, “Expertus: A generator approach to automate performance testing in IaaS clouds,” in *5th International Conference on Cloud Computing (CLOUD)*, pp. 115–122, IEEE, 2012.

[173] G. S. Swint, C. Pu, G. Jung, W. Yan, Y. Koh, Q. Wu, C. Consel, A. Sahai, and K. Moriyama, “Clearwater: extensible, flexible, modular code generation,” in *Proceedings of the 20th International Conference on Automated Software Engineering*, pp. 144–153, IEEE/ACM, 2005.

[174] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson, “Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0,” in *Proc. of CCA*, vol. 8, 2008.

[175] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the weather tomorrow? Towards a benchmark for the cloud," in *Proceedings of the 2nd International Workshop on Testing Database Systems*, p. 9, ACM, 2009.

[176] X. Wang, J. Zhang, M. Wang, L. Zu, Z. Lu, and J. Wu, "CDCAS: A Novel Cloud Data Center Security Auditing System," in *11th International Conference on Services Computing (SCC)*, pp. 605–612, IEEE, 2014.

[177] F. Doelitzscher, C. Reich, M. Knahl, and N. Clarke, "Incident detection for cloud environments," in *The 3rd International Conference on Emerging Network Intelligence (EMERGING)*, pp. 100–105, 2011.

[178] F. Doelitzscher, C. Fischer, D. Moskal, C. Reich, M. Knahl, and N. Clarke, "Validating cloud infrastructure changes by cloud audits," in *8th World Congress on Services (SERVICES)*, pp. 377–384, IEEE, 2012.

[179] A. Haeberlen, "A case for the accountable cloud," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 52–57, 2010.

[180] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable Virtual Machines," in *OSDI*, pp. 119–134, 2010.

[181] R. K. Ko, B. S. Lee, and S. Pearson, "Towards achieving accountability, auditability and trust in cloud computing," *Advances in Computing and Communications*, pp. 432–444, 2011.

[182] R. Accorsi and L. Lowis, "ComCert: Automated Certification of Cloud-based Business Processes," *ERCIM News*, vol. 2010, no. 83, pp. 50–51, 2010.

[183] A. Awad and M. Weske, "Visualization of compliance violation in business process models," in *Business process management workshops*, pp. 182–193, Springer, 2010.

[184] A. Awad, G. Decker, and M. Weske, "Efficient compliance checking using BPMN-Q and temporal logic," in *International Conference on Business Process Management*, pp. 326–341, Springer, 2008.

[185] J. Yao, S. Chen, C. Wang, D. Levy, and J. Zic, "Accountability as a service for the cloud," in *International Conference on Services Computing (SCC)*, pp. 81–88, IEEE, 2010.

[186] Z. Birnbaum, B. Liu, A. Dolgikh, Y. Chen, and V. Skormin, "Cloud Security Auditing Based on Behavioral Modeling," in *9th World Congress on Services (SERVICES)*, pp. 268–273, IEEE, 2013.

[187] J. S. Park, E. Spetka, H. Rasheed, P. Ratazzi, and K. J. Han, "Near-real-time cloud auditing for rapid response," in *26th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pp. 1252–1257, IEEE, 2012.

[188] R. Nix, M. Kantarcioglu, and S. Shetty, "Toward a Real-Time Cloud Auditing Paradigm," in *9th World Congress on Services (SERVICES)*, pp. 255–259, IEEE, 2013.

[189] M. Wang, V. Holub, T. Parsons, J. Murphy, and P. O'Sullivan, "Scalable run-time correlation engine for monitoring in a cloud computing environment," in *17th International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pp. 29–38, IEEE, 2010.

[190] S. Clayman, A. Galis, C. Chapman, G. Toffetti, L. Rodero Merino, L. M. Vaquero, K. Nagin, and B. Rochwerger, "Monitoring service clouds in the future internet," in *Future Internet Assembly*, pp. 115–126, 2010.

[191] M. Anand, "Cloud monitor: monitoring applications in cloud," in *International Conference on Cloud Computing in Emerging Markets (CCEM)*, pp. 1–4, IEEE, 2012.

[192] S. Zareian, M. Fokaefs, H. Khazaei, M. Litoiu, and X. Zhang, "A big data framework for cloud monitoring," in *Proceedings of the 2nd International Workshop on BIG Data Software Engineering*, pp. 58–64, ACM, 2016.

[193] L. Kai, T. Weiqin, Z. Liping, and H. Chao, "Scm: A design and implementation of monitoring system for cloudstack," in *International Conference on Cloud and Service Computing (CSC)*, pp. 146–151, IEEE, 2013.

[194] J. Shao, H. Wei, Q. Wang, and H. Mei, "A runtime model based monitoring approach for cloud," in *3rd international conference on Cloud Computing (CLOUD)*, pp. 313–320, IEEE, 2010.

[195] D. Zhao, "Toward Real-time and Fine-grained Monitoring of Software-defined Networking in the Cloud," in *9th International Conference on Cloud Computing (CLOUD)*, pp. 884–887, IEEE, 2016.

[196] S. Sundareswaran, A. Squicciarini, and D. Lin, "Ensuring distributed accountability for data sharing in the cloud," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 556–568, 2012.

[197] D. Zou, W. Zhang, W. Qiang, G. Xiang, L. T. Yang, H. Jin, and K. Hu, "Design and implementation of a trusted monitoring framework for cloud platforms," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2092–2102, 2013.

[198] S. A. De Chaves, R. B. Uriarte, and C. B. Westphall, "Toward an architecture for monitoring private clouds," *IEEE Communications Magazine*, vol. 49, no. 12, pp. 130–137, 2011.

[199] A. T. Monfared and M. G. Jaatun, "Monitoring intrusions and security breaches in highly distributed cloud environments," in *3rd International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 772–777, IEEE, 2011.

[200] M. T. Khorshed, A. S. Ali, and S. A. Wasimi, "Monitoring insiders activities in cloud computing using rule based learning," in *10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 757–764, IEEE, 2011.

[201] F. J. Krautheim, "Private Virtual Infrastructure for Cloud Computing," *Proceedings of the 1st USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[202] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, “Cloud monitoring: A survey,” *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[203] M. H. Mohamaddiah, A. Abdullah, S. Subramaniam, and M. Hussin, “A survey on resource allocation and monitoring in cloud computing,” *International Journal of Machine Learning and Computing*, vol. 4, no. 1, p. 31, 2014.

[204] K. Mahbub and G. Spanoudakis, “Monitoring WS-Agreements: An Event Calculus–Based Approach,” in *Test and Analysis of Web Services*, pp. 265–306, Springer, 2007.

[205] G. Spanoudakis, C. Kloukinas, and K. Mahbub, “The serenity runtime monitoring framework,” in *Security and Dependability for Ambient Intelligence*, pp. 213–237, Springer, 2009.

[206] K. Mahbub, G. Spanoudakis, and T. Tsigkritis, “Translation of SLAs into monitoring specifications,” in *Service Level Agreements for Cloud Computing*, pp. 79–101, Springer, 2011.

[207] H. Foster and G. Spanoudakis, “Advanced service monitoring configurations with SLA decomposition and selection,” in *Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 1582–1589, ACM, 2011.

[208] H. Foster and G. Spanoudakis, “SMaRT: a workbench for reporting the monitorability of services from SLAs,” in *Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems*, pp. 36–42, ACM, 2011.

[209] K. Alhamazani, R. Ranjan, K. Mitra, F. Rabhi, P. P. Jayaraman, S. U. Khan, A. Guabtni, and V. Bhatnagar, “An overview of the commercial cloud monitoring tools: Research dimensions, design issues, and state-of-the-art,” *Computing*, vol. 97, no. 4, pp. 357–377, 2015.

[210] M. L. Massie, B. N. Chun, and D. E. Culler, “The ganglia distributed monitoring system: Design, implementation, and experience,” *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[211] M. Massie, B. Li, B. Nicholes, V. Vuksan, R. Alexander, J. Buchbinder, F. Costa, A. Dean, D. Josephsen, P. Phaal, *et al.*, *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale*. O’Reilly Media, 2012.

[212] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler, “Wide area cluster monitoring with Ganglia,” in *CLUSTER*, vol. 3, pp. 289–289, 2003.

[213] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu, “Monalisa: A distributed monitoring service architecture,” *arXiv preprint cs/0306096*, 2003.

[214] I. Legrand, H. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, C. Dobre, A. Muraru, A. Costan, M. Dediu, and C. Stratan, “MonALISA: An agent based, dynamic service system to monitor, control and optimize distributed systems,” *Computer Physics Communications*, vol. 180, no. 12, pp. 2472–2498, 2009.

[215] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX conference on System administration (Lisa)*, vol. 99, pp. 229–238, 1999.

[216] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23, pp. 2435–2463, 1999.

[217] L. Chuanyi, L. Jie, and F. Binxing, "T-YUN: Trustworthiness Verification and Audit on the Cloud Providers," *IEICE TRANSACTIONS on Information and Systems*, vol. 96, no. 11, pp. 2344–2353, 2013.

[218] S. Xiang, B. Zhao, A. Yang, and T. Wei, "Dynamic measurement protocol in infrastructure as a service," *Tsinghua Science and Technology*, vol. 19, no. 5, pp. 470–477, 2014.

[219] I. Khan, H.-u. Rehman, and Z. Anwar, "Design and deployment of a trusted eucalyptus cloud," in *8th International Conference on Cloud Computing (CLOUD)*, pp. 380–387, IEEE, 2011.

[220] A. Ruan and A. Martin, "RepCloud: Attesting to Cloud Service Dependency," *IEEE Transactions on Services Computing*, 2016.

[221] A. Noman and C. Adams, "Hardware-based DLAS: Achieving geo-location guarantees for cloud data using TPM and Provable Data Possession," in *17th International Conference on Computer and Information Technology (ICCIT)*, pp. 280–285, IEEE, 2014.

[222] R. Yeluri and E. Castro Leon, "Boundary Control in the Cloud: Geo-Tagging and Asset Tagging," in *Building the Infrastructure for Cloud Security*, pp. 93–121, Springer, 2014.

[223] A. Vaish, A. Kushwaha, R. Das, and C. Sharma, "Data Location Verification in Cloud Computing," *International Journal of Computer Applications*, vol. 68, no. 12, 2013.

[224] J. Gao, X. Bai, and W.-T. Tsai, "Cloud testing-issues, challenges, needs and practice," *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 9–23, 2011.

[225] X. Bai, M. Li, B. Chen, W.-T. Tsai, and J. Gao, "Cloud testing tools," in *6th International Symposium on Service Oriented System Engineering (SOSE)*, pp. 1–12, IEEE, 2011.

[226] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[227] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: A programmable tool for multiple-failure injection," in *ACM SIGPLAN Notices*, vol. 46, pp. 171–188, ACM, 2011.

[228] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci Dusseau, R. H. Arpaci Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A framework for cloud recovery testing," in *Proceedings of 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, p. 239, 2011.

[229] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2010.

[230] C. Bennett, R. L. Grossman, D. Locke, J. Seidman, and S. Vejcik, "Malstone: Towards a benchmark for analytics on large data clouds," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 145–152, ACM, 2010.

[231] J. Gao, K. Manjula, P. Roopa, E. Sumalatha, X. Bai, W.-T. Tsai, and T. Uehara, "A cloud-based TaaS infrastructure with tools for SaaS validation, performance and scalability evaluation," in *4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 464–471, IEEE, 2012.

[232] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara, "Testing as a service (TaaS) on clouds," in *7th International Symposium on Service Oriented System Engineering (SOSE)*, pp. 212–223, IEEE, 2013.

[233] W. T. Tsai, G. Qi, L. Yu, and J. Gao, "Taas (testing-as-a-service) design for combinatorial testing," in *8th International Conference on Software Security and Reliability*, pp. 127–136, IEEE, 2014.

[234] X. Bai, M. Li, X. Huang, W.-T. Tsai, and J. Gao, "Vee@ Cloud: The virtual test lab on the cloud," in *Proceedings of the 8th International Workshop on Automation of Software Test*, pp. 15–18, IEEE, 2013.

[235] G. Candea, S. Bucur, and C. Zamfir, "Automated software testing as a service," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 155–160, ACM, 2010.

[236] L. Yu, L. Zhang, H. Xiang, Y. Su, W. Zhao, and J. Zhu, "A Framework of Testing as a Service," in *International Conference on Management and Service Science (MASS)*, pp. 1–4, IEEE, 2009.

[237] L. Yu, W.-T. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao, "Testing as a Service over Cloud," in *5th International Symposium on Service Oriented System Engineering (SOSE)*, pp. 181–188, IEEE, 2010.

[238] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[239] T. Parveen, S. Tilley, N. Daley, and P. Morales, "Towards a distributed execution framework for JUnit test cases," in *International Conference on Software Maintenance (ICSM)*, pp. 425–428, IEEE, 2009.

[240] T. Parveen and S. Tilley, "When to migrate software testing to the cloud?," in *3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pp. 424–427, IEEE, 2010.

[241] S. Baride and K. Dutta, "A cloud based software testing paradigm for mobile applications," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 3, pp. 1–4, 2011.

[242] Z. Ganon and I. E. Zilbershtein, "Cloud-based performance testing of network management systems," in *14th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pp. 1–6, IEEE, 2009.

[243] L. M. Riungu, O. Taipale, and K. Smolander, "Software testing as an online service: Observations from practice," in *3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pp. 418–423, IEEE, 2010.

[244] P. Stephanow, G. Srivastava, and J. Schütte, "Test-based cloud service certification of opportunistic providers," in *9th International Conference on Cloud Computing (CLOUD)*, pp. 843–848, IEEE, 2016.

[245] P. Stephanow and C. Banse, "Clouditor – Continuous Cloud Assurance." Available: https://www.aisec.fraunhofer.de/content/dam/aisec/Dokumente/Publikationen/Studien_TechReports/englisch/Whitepaper_Clouditor_Feb2017.pdf, Feb. 2017. [Accessed: 2018-12-13].

[246] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services," in *USENIX Security Symposium*, pp. 175–188, 2012.

[247] International Organization for Standardization (ISO), "The ISO Survey of Management System Standard Certifications 2015." Available: https://www.iso.org/files/live/sites/isoorg/files/standards/conformity_assessment/certification/doc/survey_executive-summary.pdf, 2015. [Accessed: 2018-12-13].

[248] "Amazon Web Services (AWS) ISO/IEC 27001:2013 certification." Available: https://d0.awsstatic.com/certifications/iso_27001_global_certification.pdf, Dec. 2017. [Accessed: 2018-12-13].

[249] P. Stephanow and K. Khajehmoogahi, "Towards continuous security certification of Software-as-a-Service applications using web application testing techniques," in *31th International Conference on Advanced Information Networking and Applications (AINA)*, pp. 931–938, IEEE, 2017.

[250] P. Stephanow, M. Moein, and C. Banse, "Continuous location validation of cloud service components," in *9th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 255–262, IEEE, 2017.

[251] A. S. Tanenbaum and M. Van Steen, *Distributed systems*. Prentice-Hall, 2007.

[252] "Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region." [Online]. Available: https://aws.amazon.com/message/41926/, 2017. [Accessed: 2018-12-13].

[253] B. The German Federal Office for Information Security (Bundesamt für Informationssicherheit, "Referencing Cloud Computing Compliance Controls Catalogue (C5) to International Standards." Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/CloudComputing/ComplianceControlsCatalogue/Referencing_Cloud_Computing_Compliance_Controls_Catalogue.pdf?__blob=publicationFile&v=2, 2016. [Accessed: 2018-12-13].

[254] P. Gill, Y. Ganjali, B. Wong, and D. Lie, "Dude, where's that IP? circumventing measurement-based IP geolocation," in *Proceedings of the 19th USENIX conference on Security*, pp. 16–16, USENIX Association, 2010.

[255] C. M. Bishop, "Pattern recognition," *Machine Learning*, vol. 128, pp. 1–58, 2006.

[256] R. V. Oliveira, B. Zhang, and L. Zhang, "Observing the evolution of Internet AS topology," in *ACM SIGCOMM Computer Communication Review*, vol. 37, pp. 313–324, ACM, 2007.

[257] R. Pastor Satorras, A. Vázquez, and A. Vespignani, "Dynamical and correlation properties of the Internet," *Physical review letters*, vol. 87, no. 25, p. 258701, 2001.

[258] J. C. Schlimmer and R. H. Granger, "Incremental learning from noisy data," *Machine learning*, vol. 1, no. 3, pp. 317–354, 1986.

[259] I. Zliobaite, "Learning under concept drift: An overview," *arXiv preprint arXiv:1010.4784*, 2010.

[260] "Regulation (EU) 2016/679 of the European Parliament and of the Council (General Data Protection Regulation)." Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2016.119.01.0001.01.ENG, Apr. 2016. [Accessed, 2018-12-13].

[261] "Directive 95/46/EC of the European Parliament of the Council (Data Protection Directive)." Available: http://eur-lex.europa.eu/legal-content/en/TXT/?uri=CELEX:31995L0046, Oct. 1995. [Accessed: 2018-12-13].

[262] "Australian Privacy Principles (APP)." Available: https://www.oaic.gov.au/individuals/privacy-fact-sheets/general/privacy-fact-sheet-17-australian-privacy-principles, Mar. 2014. [Accessed: 2018-12-13].

[263] R. O. Duda, P. E. Hart, D. G. Stork, *et al.*, *Pattern classification*, vol. 2. Wiley New York, 1973.

[264] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[265] V. N. Vapnik and V. Vapnik, *Statistical learning theory*, vol. 1. Wiley New York, 1998.

[266] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[267] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[268] V. Hautamaki, I. Karkkainen, and P. Franti, "Outlier detection using k-nearest neighbour graph," in *Proceedings of the 17th International Conference on Pattern Recognition (ICPR)*, vol. 3, pp. 430–433, IEEE, 2004.

[269] B. Schölkopf, J. C. Platt, J. Shawe Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural computation*, vol. 13, no. 7, pp. 1443–1471, 2001.

[270] Internet Engineering Task Force (IETF), "RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2." [Online]. Available: `https://tools.ietf.org/html/rfc5246`, Aug. 2008. [Accessed: 2018-12-13].

[271] Internet Engineering Task Force (IETF), "RFC 7465: Prohibiting RC4 Cipher Suites." [Online]. Available: `https://tools.ietf.org/html/rfc7465`, Feb. 2015. [Accessed: 2018-12-13].

[272] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, pp. 475–488, ACM, 2014.

[273] The Open Web Application Security Project (OWASP), "REST Security Cheat Sheet." [Online]. Available: `https://www.owasp.org/index.php/REST_Security_Cheat_Sheet`. [Accessed: 2018-12-13].

[274] Common Vulnerabilities and Exposures (CVE), "Compression Ratio Info-leak Made Easy (CRIME, CVE-2012-4929)." Available: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2012-4929`, Sept. 2012. [Accessed: 2018-12-13].

[275] Common Vulnerabilities and Exposures (CVE), "CCS Injection Vulnerability (CVE-2014-0224)." [Online]. Available: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0224`, Dec. 2013. [Accessed: 2018-12-13].

[276] Internet Engineering Task Force (IETF), "RFC 7507: TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks." [Online]. Available: `https://tools.ietf.org/html/rfc7507`, [Accessed: 2018-12-13].

[277] Internet Engineering Task Force (IETF), "Transport Layer Security (TLS) Renegotiation Indication Extension." [Online]. Available: `https://tools.ietf.org/html/rfc5746`. [Accessed: 2018-12-13].

[278] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st ed., 2012.

[279] B. Grobauer, T. Walloschek, and E. Stöcker, "Understanding cloud computing vulnerabilities," *IEEE Security & Privacy*, vol. 9, no. 2, pp. 50–57, 2011.

[280] The OWASP Foundation, "OWASP Top 10 - 2013: The Ten Most Critical Web Application Security Risks," *The Open Web Application Security Project*, 2013.

[281] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in *Proceedings of the International Symposium on Secure Software Engineering*, vol. 1, pp. 13–15, IEEE, 2006.

[282] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation, 3rd Edition*. Pearson, 2013.

[283] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[284] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, *DSL engineering: Designing, implementing and using domain-specific languages*. 2013. Available: `http://dslbook.org/`, [Accessed: 2018-12-13].

[285] Capers Jones, "Programming Languages Table." [Online]. Available: `http://www.cs.bsu.edu/homepages/dmz/cs697/langtbl.htm`, March 1996. [Accessed: 2018-12-13].

[286] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton, "A software engineering experiment in software component generation," in *Proceedings of the 18th International Conference on Software Engineering*, pp. 542–552, IEEE, 1996.

[287] J. Gray and G. Karsai, "An examination of DSLs for concisely representing model traversals and transformations," in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, pp. 10–20, IEEE, 2003.

[288] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[289] J. W. Backus, "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.

[290] R. E. Pattis, "Ebnf: A notation to describe syntax," 2013.

[291] N. Wirth, "Extended Backus-Naur form (EBNF)," *ISO/IEC*, 1996.

[292] W3C, "Extensible Markup Language (XML) 1.0 (Third Edition) – Section Notation: The formal grammar of XML using Extended Backus-Naur Form (EBNF) notation." [Online]. Available: `https://www.w3.org/TR/xml/#sec-notation`, 2004. [Accessed: 2018-12-13].

[293] I. Hughes and T. Hase, *Measurements and their uncertainties: A practical guide to modern error analysis*. Oxford University Press, 2010.

[294] B. N. Taylor and C. E. Kuyatt, *Guidelines for evaluating and expressing the uncertainty of NIST measurement results*. US Department of Commerce, Technology Administration, and National Institute of Standards and Technology (NIST), 1994.

[295] A. B. Owen, *Monte Carlo theory, methods and examples*. 2013. Available: http://statweb.stanford.edu/~owen/mc/, [Accessed: 2018-12-13].

[296] G. E. Box, W. G. Hunter, and J. S. Hunter, *Statistics for experimenters: An introduction to design, data analysis, and model building*, vol. 1. JSTOR, 1978.

[297] D. Freedman, R. Pisani, and R. Purves, "Statistics – 4th Edition," *W.W. Norton & Company*, 2007.

[298] D. A. Freedman, *Statistical models: Theory and practice*. Cambridge University Press, 2009.

[299] T. Fawcett, "An introduction to ROC analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[300] D. M. Powers, *Evaluation: From Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation*. Bioinfo Publications, 2011.

[301] S. V. Stehman, "Selecting and interpreting measures of thematic classification accuracy," *Remote Sensing of Environment*, vol. 62, no. 1, pp. 77–89, 1997.

[302] A. C. Yao, "Protocols for secure computations," in *23rd Annual Symposium on Foundations of Computer Science (SFCS)*, pp. 160–164, IEEE, 1982.

[303] O. Goldreich, S. Micali, and A. Wigderson, "How to solve any protocol problem," in *Proceedings of the 19th STOC*, pp. 218–229, 1987.

[304] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT press Cambridge, 2008.

Acronyms

API Application Programming Interface

APP Australian Privacy Principles

AS Autonomous System

ASCT Automated Security Compliance Tool

AST Abstract Syntax Tree

AUDITOR European Cloud Service Data Protection Certification

AVM Accountable Virtual Machines

AWS Amazon Web Service

BGP Border Gateway Protocol

BMWi Bundesministerium für Wirtschaft und Energie

BNF Backus-Naur Form

BSI Bundesamt für Sicherheit in der Informationstechnik

CBDG Constraint-Based Data Geolocation

CBG Constraint-Based Geolocation

CC Common Criteria

CCM Cloud Control Matrix

CCSL Cloud Certification Schemes List

CDCAS Cloud Data Center Security Auditing System

CISSP Certified Information Systems Security Professional

CMM Capability Maturity Model

CMMI Capability Maturity Model Integration

CMU Carnegie Mellon University

CPU Central Processing Unit

CSA Cloud Security Alliance

CUMULUS Certification Infrastructure for Multi-Layer Cloud Services

DNS Domain Name System

DSL Domain-specific Language

EBNF Extended Backus-Naur Form

EC2 Amazon Elastic Compute Cloud

EU European Union

EU-SEC European Security Certification Framework

FedRamp Federal Risk and Authorization Management Program

GDPR General Data Protection Regulation

GPL General Purpose Language

IaaS Infrastructure-as-a-Service

IAF Information Assurance Framework

ICMP Internet Control Message Protocol

IEC International Electrotechnical Commission

IP Internet Protocol

ISO International Organization for Standardization

LinearSVC Linear Support Vector Clustering

NIST National Institute of Standards and Technology

NGCert Next Generation Certification

PaaS Platform-as-a-Service

PoR Proof of Retrievability

QoS Quality of Service

SAaaS Security Audit as a Service

SaaS Software-as-a-Service

SEI Software Engineering Institute

SLA Service Level Agreement

SP Special Publication

SPICE Software Process Improvement and Capability Determination

SSH Secure Shell

SUT Service under Test

SVM Support Vector Machine

TaaS Testing as a Service

TBG Topology-Based Geolocation

TCP Transmission Control Protocol

TCRR TPM-based Certification of a Remote Resource

TOE Target of Evaluation

TPM Trusted Platform Module

VAT Vulnerability Assessment Tools

VCS Virtual Coordinate Systems

VFDT Very Fast Decision Tree

VM Virtual Machine

VMI Virtual Machine Introspection

vTPM virtualized Trusted Platform Module

Appendix A

External tool configurations

Parameters used with sklearn.svm.LinearSVC Listing A.1 shows the values passed to parameters of *LinearSVC* within the implementation of the example test scenario *Continuously testing location* presented in Section 5.3. LinearSVC is a method of the *Scikit-learn* library[121].

Listing A.1: Configuration of LinearSVC

```
1
2  LinearSVC(C=1.0,
3            class_weight=None,
4            dual=True,
5            fit_intercept=True,
6            intercept_scaling=1,
7            loss='squared_hinge',
8            max_iter=1000,
9            multi_class='ovr',
10           penalty='l2',
11           random_state=None,
12           tol=0.0001,
13           verbose=0)
```

Parameters used with sklearn.svm.OneClassSVM Listing A.2 shows the values passed to parameters of *OneClassSVM* within the implementation of the example test scenario *Continuously testing location* presented in Section 5.3. OneClassSVM is a method of the *Scikit-learn* library[122].

Listing A.2: Configuration of OneClassSVM

```
1
2  OneClassSVM(cache_size=200,
3              coef0=0.0,
4              degree=3,
5              gamma='auto',
6              kernel='linear',
7              max_iter=-1,
8              nu=0.01,
9              random_state=None,
10             shrinking=True,
11             tol=0.001,
12             verbose=False)
```

[121] http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html [Accessed: 2018-12-13]

[122] http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html [Accessed: 2018-12-13]

Appendix B

Code snippets

XTend Code generator to translate ConTest to YAML Listing B.1 contains the code of the XTend class *ConTestDSLGenerator* whose *compile* method translates ConTest to YAML. Note that line breaks from lines 71 to 72 of Listing B.1 are added for enhanced readability and have to be removed to correctly generate the YAML file shown in Listing 6.6, i.e., the code in line 72 has to be appended to line 71.

Listing B.1: ConTestDSLGenerator class

```

package de.fraunhofer.aisec.conTestDSL.generator

import com.google.inject.Inject
import de.fraunhofer.aisec.conTestDSL.conTestDSL.KeyValue
import de.fraunhofer.aisec.conTestDSL.conTestDSL.Test
import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import org.eclipse.xtext.naming.IQualifiedNameProvider

class ConTestDSLGenerator extends AbstractGenerator {

    @Inject extension IQualifiedNameProvider

  override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
    for (e : resource.allContents.toIterable.filter(Test)) {
      fsa.generateFile(e.fullyQualifiedName.toString("/") + ".yaml",
        e.compile
      )
    }

  }

     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»'class': «tc.testCaseModule»
        «length4»order: «tc.order»
        «IF tc.inputParams !==null»
             «FOR ip : tc.inputParams»
             «FOR kv : ip.params»
          «length4»«kv.key»: «identifyParams(kv)»
                «ENDFOR»
                 «ENDFOR»
```

```
53            «ENDIF»
54            «FOR ap : tc.assertParams»
55            «FOR kv : ap.params»
56              «length4»«kv.key»: «identifyParams(kv)»
57            «ENDFOR»
58            «ENDFOR»
59
60            «ENDFOR»
61            workflow:
62            «length2»class: «ct.workflow.workflowModule»
63            «length2»name: «ct.workflow.workflowName»
64            «length2»testSuites:
65            «FOR ts : ct.testSuites»
66            «length4»«ts.name»:
67            «length4»name: «ts.name»
68            «length4»label: «ts.testSuiteName»
69            «length4»randomized: «IF ts.randInterval !== null»true «ELSE»false«ENDIF»
70            «length4»iteration: «IF ts.iteration.infinite!==null»-1«ELSE»«ts.iteration.count»«ENDIF»
71            «length4»interval: «IF ts.randInterval !== null»[«ts.randInterval.leftBound»,«ts.randInterval.rightBound»]
72            «ELSE»«IF ts.seqFixedInterval !==null»«ts.seqFixedInterval.elements»«ELSE»[«ts.fixedInterval»]«ENDIF»«ENDIF»
73            «length4»offset: «ts.off»
74            «length4»timeout: «ts.timeout»
75            «length4»testCases: [«FOR tc:ct.testCases SEPARATOR ','»'@ref': «tc.name»«ENDFOR»]
76            «ENDFOR»
77        '''
78      def identifyParams(KeyValue kv){
79          switch kv{
80            case kv.listInt !== null: kv.listInt.elements
81            case kv.listString !== null: kv.listString.elements
82            case kv.stringVal !== null: kv.stringVal
83            default : kv.intValue
84            }
85      }
86  }
```

Appendix C

Publications in the context of this thesis

Hereafter, we list all peer-reviewed publications which have been published in the course of this thesis. The contents of these publications either present intermediate results of this thesis or are related to the research in context of continuous test-based cloud service certification. We outline how these publications are related to this thesis.

- **Philipp Stephanow and Mark Gall.** ***Language Classes for Cloud Service Certification Systems***. Proceedings of the 11th IEEE World Congress on Services (SERVICES), pp. 127–134, New York, 2015.

  Relation to thesis This paper introduces language classes for cloud service certification systems to facilitate research in design and implementation of these systems. This work is a preliminary effort for the domain-specific language *ConTest* which is presented in Chapter 6 of this thesis. The purpose of ConTest is to rigorously define continuous tests.

- **Philipp Stephanow and Niels Fallenbeck.** ***Towards Continuous Certification of Infrastructure-as-a-Service Using Low-Level Metrics***. Proceedings of the 12th IEEE International Conference on Ubiquitous Intelligence and Computing, 12th IEEE International Conference on Autonomic and Trusted Computing and 15th IEEE International Conference on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), pp. 1485–1492, Beijing, 2015.

  Relation to thesis This paper investigates low-level metrics which are provided by widely deployed implementations of components involved in delivery of Infrastructure-as-a-Service (IaaS). This work is mainly a complementary effort to this thesis since it analyses evidence available to monitoring-based certification models. The insights obtained within this publication contribute to the background (Chapter 2) as well as to the related work (Chapter 3) of this thesis.

- **Philipp Stephanow, Christian Banse, Julian Schütte.** ***Generating Threat Profiles for Cloud Service Certification Systems***. Proceedings of the 17th IEEE High Assurance Systems Engineering Symposium (HASE), pp. 260-267, Orlando, 2016.

  Relation to thesis This paper proposes an approach to model architecture variants of cloud service certification systems and analyzes corresponding threats these systems

face. This work contributes to the discussion of our framework to design continuous tests presented in Chapter 4 of this thesis. Furthermore, we use the results of this paper to identify future research directions in the conclusion of this thesis (Chapter 9).

- **Philipp Stephanow, Gaurav Srivastava, and Julian Schütte.** ***Test-based Cloud Service Certification of Opportunistic Providers***. Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD), pp. 843-848, San Francisco, 2016.

  Relation to thesis This paper present an approach to support test-based cloud service certification of opportunistic cloud service providers. The contents of this paper can be understood as a summary of Chapter 8 which introduces a method to model and reason about the behavior of an opportunistic cloud service provider and points out countermeasures through randomization which are also integrated into our framework to design continuous tests. This paper also includes an outline of the building blocks of our framework to design tests presented in Chapter 4 of this thesis. Further, we reuse parts of the implementation section of this paper within Chapter 5 which contains the example continuous test scenarios.

- **Immanuel Kunz and Philipp Stephanow.** ***A Process Model to Support Continuous Certification of Cloud Services***. Proceedings of the 31th IEEE International Conference on Advanced Information Networking and Applications (AINA), pp. 986-993, Taipei, 2017.

  Relation to thesis This paper analyzes and generalizes the traditional certification processes and, on this basis, develops a novel, executable process model to support continuous cloud service certification. The results presented in this paper are mainly complementary to this thesis since they provide means to integrate continuous tests into existing certification processes as well as update continuous tests during operation.

- **Philipp Stephanow and Koosha Khajehmoogahi.** ***Towards Continuous Security Certification of Software-as-a-Service Applications Using Web Application Testing Techniques***. Proceedings of the 31th IEEE International Conference on Advanced Information Networking and Applications (AINA), pp. 931-938, Taipei, 2017.

  Relation to thesis This paper reports on intermediate results of developing methods as well as tools to support continuous test-based security certification of Software-as-a-Service (SaaS) applications. This work is partly reused in thesis as one of the example continuous test scenarios presented in Chapter 5. Furthermore, we generalize some of the research challenges brought forward within this paper and used them in the Introduction of this thesis (Chapter 1).

- **Philipp Stephanow and Christian Banse.** ***Evaluating the Performance of Continuous Test-based Cloud Service Certification***. Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 1117–1126, Madrid, 2017.

  Relation to thesis This paper evaluates the performance of test-based cloud service certification techniques when they are executed continuously. This paper introduces

the universal test metrics which are described in Chapter 4 of this thesis. Also, the contents of this paper can be understood as a summary of Chapter 7 of this thesis which introduces a method to evaluate the accuracy and precision of tests which support continuous cloud service certification.

- **Philipp Stephanow, Mohammad Moein and Christian Banse.** ***Continuous Location Validation of Cloud Service Components***. Proceedings of the 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 255-262, Hong Kong, 2017.

  Relation to thesis This paper presents adaptive location classification, an approach to continuously validate the location of cloud service components. The underlying method of this work is used to describe the example test scenario *Continuously testing location* which is presented in Section 5.3 of Chapter 5. Furthermore, the experimental results delineated in the paper can be understood as an extract of the results discussed in Section 5.3.4.