Modeling Time, Probability, and Configuration Constraints for Continuous Cloud Service Certification

M. Anisetti[c], C.A. Ardagna[c], E. Damiani[b], N. El Ioini[a], F. Gaudenzi[c]

[a] *Free University of Bozen, Bolzano, Italy*
[b] *Etisalat British Telecom Innovation Center, Khalifa University of Science, Technology and Research, Abu Dhabi, UAE*
[c] *Dipartimento di Informatica, Università degli Studi di Milano, Milano, Italy*

Abstract

Cloud computing proposes a paradigm shift where resources and services are allocated, provisioned, and accessed at runtime and on demand. New business opportunities emerge for service providers and their customers, at a price of an increased uncertainty on how their data are managed and their applications operate once stored/deployed in the cloud. This scenario calls for assurance solutions that formally assess the working of the cloud and its services/processes. Current assurance techniques increasingly rely on model-based verification, but fall short to provide sound checks on the validity and correctness of their assessment over time. The approach in this paper aims to close this gap catching unexpected behaviors emerging when a verified service is deployed in the target cloud. We focus on certification-based assurance techniques, which provide customers with verifiable and formal evidence on the behavior of cloud services/processes. We present a trustworthy cloud certification scheme based on the continuous verification of model correctness against real and synthetic service execution traces, according to time, probability, and configuration constraints, and attack flows. We test the effectiveness of our approach in a real scenario involving ATOS SA eHealth application deployed on top of open source IaaS OpenStack.

Keywords: Assurance, Certification, Cloud, Compliance, Security

1. Introduction

Cloud computing has radically transformed the way in which software is procured and provisioned. Cloud-based services are becoming the primary choice for many purchasers of software products, since they offer substantial advantages in terms of efficiency, functionality, and ease of use (Gai et al. (2016, 2017)).

Email addresses: marco.anisetti@unimi.it (M. Anisetti), claudio.ardagna@unimi.it (C.A. Ardagna), ernesto.damiani@kustar.ac.ae (E. Damiani), nelioini@unibz.it (N. El Ioini), filippo.gaudenzi@unimi.it (F. Gaudenzi)

Indeed, cloud computing provides an opportunity to re-assess traditional license and tender-based software procurement strategies to create a more flexible provisioning process. Unfortunately, lack of trust in the cloud environment is partially preventing both customers and providers from taking full advantage of this opportunity (Alford and Morton (2009)). Their decision-making is in fact affected by the difficulty of evaluating risks threatening data and applications once stored/deployed in the cloud (Bellandi et al. (2015)).

To alleviate this problem, the research community has introduced new assurance techniques to enhance cloud trust and transparency, supporting different practices such as audit, certification, and compliance checking (e.g., Anisetti et al. (2013a); Bertholon et al. (2011); Doelitzscher et al. (2013); Ye et al. (2012); Sunyaev and Schneider (2013); Nunez et al. (2016)). In particular, certification has turned out to be an attractive solution for increasing the confidence of users and providers that their data and applications will be handled as agreed. Certification is evolving following IT evolution and moving from software certification (e.g., Herrmann (2002)) to service and distributed system certification (e.g., Anisetti et al. (2013a); Bertholon et al. (2011); Kourtesis et al. (2010)), and, in the last few years, to cloud-based system and infrastructure certification (e.g., Sunyaev and Schneider (2013); Anisetti et al. (2014a); Spanoudakis et al. (2012); Grobauer et al. (2011)). Today, cloud certification schemes (e.g., Anisetti et al. (2014a); Spanoudakis et al. (2012)) implement processes that can be executed at all layers of the cloud stack. Still, the dynamic nature of the cloud may impair the certification results. Certified cloud services are in fact subject to changes when deployed in the in-production environment,[1] and can become very different from their counterparts certified in a lab environment. For this reason, interest in dynamic certification is growing, and evidence collection at the basis of cloud system certification has become a continuous, runtime process often driven by a model of the system behavior (Anisetti et al. (2013a); Ravindran (2013); Spanoudakis et al. (2012)). The need for model-based, continuous evidence collection is particularly important when verifying behavioral properties that could be affected by the production execution context, such as worst-case execution times (WCET) and the actual frequency of execution paths labeled as exceptional at design time (Bate et al. (2002)). Also, the correctness of the collection process strongly relies on the correctness of the system model that if not preserved would impair the verification results and the certification process.

In this paper, we depart from the assumption of having a correct and error-free modeling of the system under certification (Anisetti et al. (2013a); Ravindran (2013); Spanoudakis et al. (2012)), and present a trustworthy cloud certification approach based on continuous verification of model correctness. Our approach aims to catch unexpected behavior occurring when a certified service is deployed in the production cloud. It builds on testing and monitoring of execution traces to discover discrepancies between the model originally used to

[1] For clarity, in the following, we refer to in-production system, environment, evaluation, as production system, environment, evaluation.

verify and certify a service in a lab environment, and the one inferred by observing system behavior in the production environment. Such discrepancies would in fact invalidate the results of a certification process, and in turn the certificate itself. We start by verifying the structure of the model, matching model paths with real and synthetic service execution traces observed in production. We then verify additional constraints, including time relations between different states of the model, transition probabilities, and mandatory configurations for certificate validity. We finally verify robustness against attacks represented in the system model.

The contribution of this paper is twofold. We first define a trustworthy certification process based on continuous verification of the correctness of the system model driving certification activities. In particular, we put forward the idea that a trustworthy certification in the cloud must be built on a verifiably correct service model capable of expressing time, probability, configuration constraints, and of specifying attack paths. Model correctness is in fact a mandatory requisite on the consistency of the certification process once the certified system is deployed in the production environment. We then design and develop a certification framework implementing a certification scheme and a methodology supporting the verification of model correctness. We note that, although our methodology is applied to a specific certification scheme, it is general and can be used for any model-based assurance technique. This paper develops on our previous work (Anisetti et al. (2014b, 2015)) by providing *i)* a trustworthy certification scheme extended with checks on model correctness (Section 2); *ii)* a methodology composed of four processes and corresponding algorithms for verifying the model structure, as well as time, probability, and configuration constraints (Section 3); *iii)* an architecture including components supporting online verification of model correctness (Section 4); *iv)* an extensive evaluation of our trustworthy certification scheme (Section 5). The evaluation considers the eHealth application of ATOS SA, a major IT enterprise, deployed on top of open source IaaS OpenStack.

The remainder of the paper is organized as follows. Section 2 sketches our trustworthy certification process. Section 3 describes the methodology for verification of model correctness at the basis of the certification process. Section 4 presents our certification framework. Section 5 presents the evaluation of the methodology and framework in a real-world scenario. Section 6 presents related work and Section 7 draws our conclusions. Finally, Appendix A-Appendix E describe the algorithms implemented in the framework for verification of model correctness.

2. Trustworthy Certification Process

A certification scheme defines an evaluation process driven by a *property*, taken from a shared vocabulary (Anisetti et al. (2013a)), to be certifiably held by a specific system representing the Target of Certification (ToC). The evaluation is usually carried out in a controlled environment (laboratory) and is aimed

at verifying the property through evidence collected by testing and/or monitoring the ToC behavior. When the ToC is executed in a dynamically changing environment like the cloud, an advanced certification framework needs to keep re-evaluating the ToC without the continuous supervision of the certification authority responsible for the certification process or the accredited lab (delegated by the certification authority) responsible for evidence collection.

Most of current techniques for cloud certification (e.g., Anisetti et al. (2014a); Lins et al. (2016b); Stephanow et al. (2016)) implement a two-step evidence collection process involving a certification authority, an accredited lab, and the system provider. The first step (*pre-deployment evaluation*) mimics traditional certification schemes and is aimed at collecting evidence in a controlled environment; the second step (*production evaluation*) addresses cloud peculiarities implementing continuous cloud-aware evaluation. In the pre-deployment evaluation, evidence collection activities are managed according to a model of the ToC (*evidence collection model*) representing only those parts relevant for property certification. Such activities are executed in a laboratory environment. The certification authority evaluates whether the collected evidence is sufficient to prove the given property and then issues a certificate for the *ToC*. We note that evidence collection model is derived from the model of the whole system including the ToC (*system model*). We also note that, though the behavior of the laboratory environment is similar to the one of the production environment, it cannot fully reproduce it. At this stage, all the processes are under the direct control of the certification authority or accredited lab executing the certification framework. In the production evaluation, possible inconsistencies (in terms of evidence) between the behavior of the ToC in laboratory and production environments are identified, and their impact on awarded certificates are evaluated. It is a continuous evaluation completely under the control of the certification process verifying the validity of the issued certificates.

There is however a subtle but important factor to be considered when production evaluation is concerned. The verification of the correctness of the ToC behavior is strictly dependent on the correctness of the evidence collection model used for managing evaluation activities, and in turn of the system model used to produce it. An error in the system model in fact can affect the results of the whole certification process. Current processes do not consider this aspect, substantially reducing the trustworthiness of the certification process. Let us consider a scenario where *i)* a ToC is certified for property *data confidentiality*, *ii)* the system model does not model a backdoor added at deployment time giving full and unrestricted access to the production environment's administrators (Duncan et al. (2012)). In this case, the production evaluation simply checking the consistency between the ToC behavior in laboratory and production environments does not uncover the inconsistency introduced by the admin's backdoor. If exposed, this backdoor would affect the validity of the certificate for property data confidentiality, possibly resulting in a certificate revocation.

To fill in this gap, we extend our model-based certification process in Anisetti et al. (2014a, 2015) with a methodology for the verification of the correctness of the system model. We assume the system model to be either manually gener-

ated by the system provider with the help of the accredited lab or automatically generated according to existing solutions (e.g., Anisetti et al. (2014b); Merten et al. (2012); Ernst et al. (1999)). Our certification process implements the pre-deployment and production evaluation steps, and extends the system model adding time, probability, configuration constraints, and attack paths tested on the ToC. These extensions represent certification meta-requirements on the ToC execution context, indirectly affecting the support of a given property, and complement traditional certification requirements on ToC mechanisms. The nature of the collected evidence is then of two kinds: *i)* evidence on the behavior of the ToC retrieved by testing/monitoring it according to the evidence collection model, *ii)* evidence on system correctness verifying the consistency between the original system model and the one retrieved by monitoring ToC executions in the production environment.

A trustworthy certification process must first verify the behavior of the ToC in production by re-executing testing/monitoring activities done in the laboratory environment and matching their results against results retrieved in the pre-deployment evaluation. It must then check the correctness of the system model used for pre-deployment certification against real and synthetic service execution traces, according to time, probability, and configuration constraints. It must finally check robustness of the ToC against modeled attack paths. Below, we assume that the pre-deployment phase results in the release of a certificate (Anisetti et al. (2014a)), and provide an approach for system model correctness verification against time, probability, and configuration constraints, and attack paths (Section 3).

3. Verification of Model Correctness

Our methodology for verifying the system (ToC) model correctness starts by matching the system model structure against production execution traces. It then analyzes additional contextual constraints on time, probability, configuration, and attack paths. Constraints and attack paths can be imposed over the system model depending on the property to be verified. It includes five verification steps as follows.

3.1. Model Structure

Model structure verification permits to catch unexpected system behaviors, which expose issues to be raised at certification authority level and eventually lead to certificate revocation. For instance, a backdoor added at deployment time (Duncan et al. (2012)) and therefore not included in the system model could affect the certificate of property confidentiality. Model structure verification does not target any specific property. Rather, it targets model correctness and is a prerequisite for the validity of the model and in turn of the certification process.

We start from the definition of the system model m representing the execution paths of a ToC composed of a specific service or a business process involving multiple services. We model m as a finite state machine as follows.

Definition 3.1 (Model m). *Model $m \in M$ is a 5-tuple (L, l_0, A, E, F), where L is a finite set of states, A is a finite set of actions (input), $E: L \times A \to L$ is the transition function, $l_0 \in L$ is the initial state, and $F \subseteq L$ is the set of final states.*

We note that a state $l \in L$ represents a specific execution point reached by the service; an action $a \in A$ represents a service call or an operation triggering a transition between two states; when clear from the context, we will refer to a transition as a pair (l_i, l_j) of states. Our model m can also be seen as a collection of execution paths formally defined as follows.

Definition 3.2 (Path p_i). *Given a service model m, a path p_i is a sequence of states $p_i = \langle l_0, \ldots, l_n \rangle$, with $l_0 \in L$ and $l_n \in F$ denoting the initial state and a final state, respectively, s.t. $\forall_{j=0}^{n-1} l_j$, $\exists$ a transition $(l_j \times a_j \to l_{j+1}) \in E$.*

Our approach checks model structure correctness by collecting execution traces from ToC executions and by projecting them on the system model paths. Execution traces are collected by either monitoring real executions of the system involving real customers or observing the results of ad hoc testing (synthetic traces executed in laboratory and/or in production). Traces can be formally defined as follows.

Definition 3.3 (Trace T_i). *An execution trace $T_i \in \mathcal{T}$ is a sequence $\langle a_1, \ldots, a_n \rangle$ of actions, where a_j is a service/operation execution.*

Appendix A illustrates the pseudocode of our algorithm for model structure verification (*MSV*). It takes as input the system model $m = (L, l_0, A, E, F)$ and collected execution traces T_i, and produces as output either *success* (1), if the traces conform to the system model, or *failure* (0), otherwise, with a description of the type of inconsistency found. *MSV* is based on a *consistency function* $\equiv$ between collected traces and system model execution (the formal definition in Appendix A). The consistency function $\equiv$ takes as input a trace T_i and a path p_j and checks if for each action in trace T_i exists a corresponding action in path p_j referring to the same service/operation. The inconsistencies can be classified according to three classes, each resulting in a set of pairs (T_i, p_j) modeling inconsistencies, as follows.

- *Partial path discovery.* A trace is consistent with a sub-path in the model meaning that, while mapping a trace to paths in the model, only an incomplete path is found. Inconsistencies of this class are stored in the set $r_{partial} = \{(T_i, p_j)\}$.
- *New path discovery.* A trace is not consistent with any path in the model meaning that a new path is found, that is, at least a new transition and/or a new state is found in the traces. Inconsistencies of this class are stored in the set $r_{new} = \{(T_i, -)\}$.
- *Broken existing path.* Real traces do not cover a path in the model and the synthetic traces return an error for the same path, meaning that the

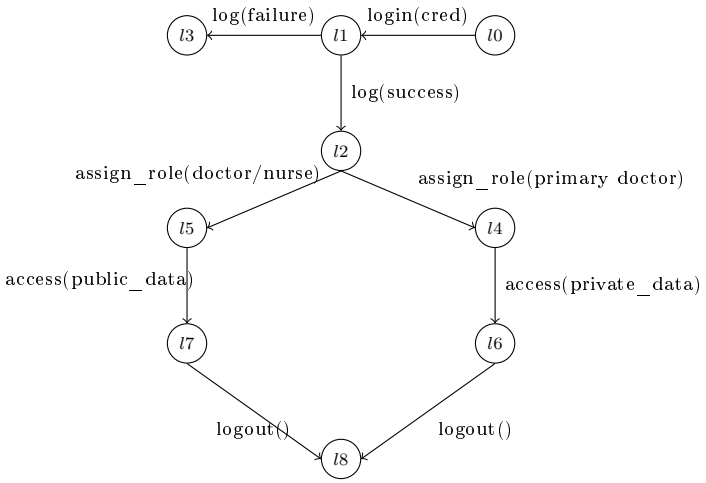


Figure 1: Model structure of the eHealth service

model includes a path that is not available/implemented in the real ToC. Inconsistencies of this class are stored in the set r_{broken}=$\{(-,p_j)\}$.

We note that r=$r_{partial}$∪r_{new}∪r_{broken} establish the basis for verifying the consistency between observed traces and paths.

Example 3.1. *Let us consider the authentication service of an eHealth system. The service checks the user's credentials and, if valid, assigns a role to her to access different portions of the eHealth system. Figure 1 shows a model of the authentication service. Once the role of the user is identified, the user has access to patients' records with different clearance levels (e.g., doctors in charge of a patient have rights to access her private data, while all doctors and nurses can access public data). Examples of traces that match this model include* $\langle login(cred),\ log(failure)\rangle$, $\langle login(cred),\ log(success),\ assign\_role(doctor),\ access(public\_data),\ logout()\rangle$. *Similarly, traces that do not match this model include* $\langle log(success),\ assign\_role(nurse),\ access(public\_data)\rangle$ *and* $\langle login(cred), log(success), assign\_role(nurse), access(private\_data), logout()\rangle$. *There is however a subtlety to consider for the latter example. In case of emergency, access to private data by doctors and nurses must be granted (*break the glass *– BG); trace* $\langle login(cred), log(success), assign\_role(nurse), access(private\_data), logout()\rangle$ *can then be observed in a production environment. If this scenario is not correctly modeled (like in Figure 1) an error is raised by MSV.*

3.2. Time Constraints

Time constraint verification permits to identify inconsistencies in the orchestration of different (parts of) services composing the ToC, when deployed in the

production environment. For instance, the verification of a given non-functional property should come with a time constraint that imposes to log any exceptional situation and/or notify involved entities within a specific time frame, not to be exceeded by the service WCET. We note that time constraints can also relate to security aspects like in the case of Distributed Denial of Service (DDoS) or latency-based attacks (Ardagna and Damiani (2014)). We use annotations on the system model in Definition 3.1 as a way to represent time constraints regulating the support of a given property. A time constraint is a set of integer-valued $x, y, \ldots$, representing clocks, and functions in the form $f(z_1,\ldots,z_i)$, representing clock functions added as annotations over the transitions of the model m. $\Phi(K)$ denotes the set of clock constraints. $\lambda_t{:}E{\rightarrow}\Phi(K)$ is a labeling function that associates a clock constraint $\phi{\in}\Phi(\mathrm{K})$ with a transition $(l_i,l_j){\in}E$ of m. Based on λ_t, we define a timed system model m_t as an extension of the system model m in Definition 3.1.

Definition 3.4 (Timed system model (m_t)). *Model m_t is a 6-tuple $(L, l_0, A, E, F, \lambda_t)$, where L, A, E, l_0, and F are defined in Definition 3.1, and λ_t assigns a label $\lambda_t((l_i,l_j))$, corresponding to the time constraint defined for each transition (l_i,l_j).*

We note that in case a transition $(l_i,l_j){\notin}E$, $\lambda_t((l_i,l_j))$ refers to the path(s) connecting $l_i{\in}L$ and $l_j{\in}L$.

To match execution traces against model m_t, traces need to be extended to support temporal properties, in other words, timestamps need to be added to each action in the trace. Formally, timed traces are defined as follows.

Definition 3.5 (Timed Trace TT_i). *A timed trace TT_i is a sequence $\langle a_1,\ldots,a_n\rangle$ of actions, where $a_j{=}\{o, t_s\}$, o is a service/operation execution, and t_s is the timestamp associated to o. The sequence of traces is temporally ordered $a_i{\prec}a_j$ meaning that a_i is temporally completed before a_j.*

Appendix B illustrates the pseudocode of our algorithm for time constraint verification (TCV). It takes as input timed system model m_t and the timed traces TT_i, and produces as output either *success* (1), if each trace satisfies time constraints in the corresponding path, or *failure* (0), otherwise, with the list of transitions violating time constraints. TCV is based on a *time consistency function* $\equiv_t$ (the formal definition in Appendix B). The consistency function $\equiv_t$ takes as input a timed trace TT_i and a path p_j, such that $TT_i{\equiv}p$, and checks if the corresponding time constraints $\lambda_t((l_i,l_j))$ annotated over path p_j are consistent with the timestamps in trace TT_i (denoted $TT{\equiv_t}p$).

Example 3.2. *We extend Example 3.1 by adding temporal guards over some of its transitions (Figure 2). For the purpose of our example we have added the transition* (BG) *in Example 3.1 referring to an emergency situation in which all doctors and nurses can access private data. In this example, we specify that in normal circumstances data access can take up to* $10ms$*, while in* BG *data access can take up to* $5ms$*. Let us consider a valid trace* $T{=}\langle login(cred),$

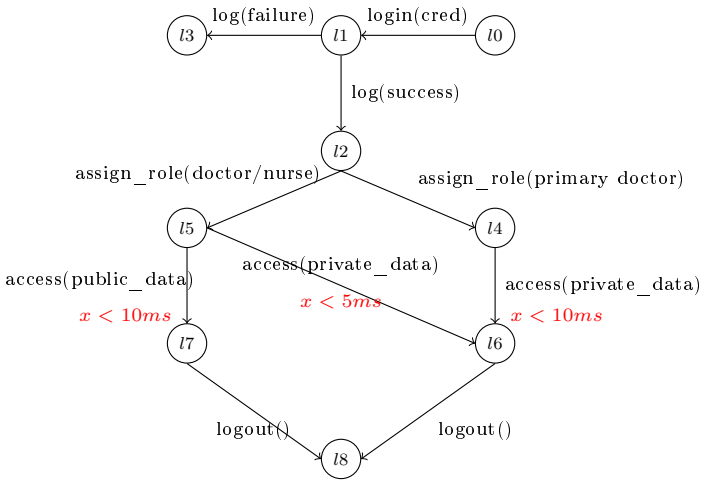


Figure 2: Time constraints of the eHealth service.

log(success), assign_role(nurse), access(private_data), logout()⟩ in m; this trace will be valid in m_t if it satisfies time constraints. For instance, timed trace TT=⟨(login(cred),2), (log(success),3), (assign_role(nurse),5), (access (private_data),8), (logout(),9)⟩ is valid because it satisfies time constraints in m_t (i.e., 8ms−5ms<5ms).

3.3. Probability Constraints

Probability constraint verification permits to identify inconsistencies in the distribution of executions among ToC flows. It considers a specific time window (consisting of a set of real and synthetic traces) and is used to identify behavioral anomalies, where exceptional activities become the norm. We use annotations on the system model in Definition 3.4 as a way to represent probability constraints regulating the support of a given property. A probability constraint is an equation of the form $x{\sim}n$, where x and n are real numbers $\in[0\dots1]$, and $\sim\in\{\leqslant,<,=,>,\geqslant\}$. $\Gamma(n)$ denotes the set of probability constraints.

Similar to time annotations, we define a labeling function $\lambda_{prob}{:}E{\rightarrow}\Gamma(n)$ that associates a probability constraint $\gamma{\in}\Gamma(n)$ with a transition $(l_i,l_{i+1}){\in}E$ of m_t. Based on λ_{prob}, we define a probabilistic timed system model m_{prob} as an extension of the system model m_t in Definition 3.4.

Definition 3.6 (Probabilistic timed system Model m_{prob}). *Model m_{prob} is a 7-tuple $(L, l_0, A, E, F, \lambda_t, \lambda_{prob})$, where L, A, E, l_0, F, and λ_t are defined in Definition 3.4, and λ_{prob} assigns a label $\lambda_{prob}((l_i,l_{i+1}))$, corresponding to the probability constraint defined for each transition (l_i,l_{i+1}).*

We note that, for each $l{\in}L$, $\sum_k\lambda_{prob}((l,l_k))=1$, with $(l,l_k){\in}E$. Let us consider a time window Δt defined as a bounded interval $[t_s,t_e]$ where $t_s{\in}\mathbb{N}$ is the

starting time, and $t_e \in \mathbb{N}$ is the ending time. The verification of model m_{prob} is built on a frequency-based evaluation of request distribution, using the subset of timed traces TT_i observed during the time window Δt.

Appendix C illustrates the pseudocode of our algorithm for probability constraint verification (PCV). It takes as input the probabilistic timed system model m_{prob} and the set $\mathcal{T}$ of timed traces TT_i, and produces as output either *success* (1), if each trace satisfies probability constraints of the corresponding path, or *failure* (0), otherwise, with the list of transitions violating probability constraints. PCV is based on a *probability consistency function* $\equiv_{prob}$ (the formal definition in Appendix C). The probability consistency function $\equiv_{prob}$ takes as input a path p and a set of timed traces TT_j, such that $TT_j \equiv p$, observed in a time window Δt and insisting on p, and checks if timed traces TT_j are such that the frequency of executing p is consistent with $\lambda_{prob}((l_i,l_{i+1}))$ annotated over p (denoted $\{TT_j\} \equiv_{prob} p$) .

Failures in the probability constraint verification means that there is an inconsistency between the probability model and the execution traces. The inconsistencies at this level show that there are changes in the execution path distribution. Thus, the system needs to be checked to verify whether the changes are due to malicious behaviors, therefore requiring the intervention of the certification authority.

Example 3.3. *We extend Example 3.2 by adding probabilities over its transitions (Figure 3). The transitions of interest include the ones representing the* BG *scenario.* BG *is a relatively rare situation that needs to be modeled with low execution probability. Thus, for a specific time window, the traces that violate the probability constraints are the ones containing transitions occurring with a percentage higher/lower than the one specified over the model m_{prob}. For instance, in the model defined in Figure 3, if for a specific time window Δt we have* 100 *valid timed traces that pass through transition (l_2,l_5), at most* .02 *of traces (2 traces) should take transition (l_5,l_6).*

3.4. Configuration Constraints

Configuration constraint verification permits to identify indirect dependencies on cloud configurations supporting the property. For instance, let us consider a system implementing a DNS made of three independent replicas. The system needs to be certified against property "reliability to two failures". At certification time, we can verify that the three replicas are actually working; however, this check does not take into account the fact that the three replicas may be deployed on the same physical hardware. In this case, the certificate for property reliability should be invalidated because if the physical machine fails, all replicas will fail. Being able to add configuration constraints as part of the service model permits to have a clear picture of the configurations that may affect the certified system, and need to be managed during the certification process as mandated by ISACA (Chaudhuri et al. (2011)).

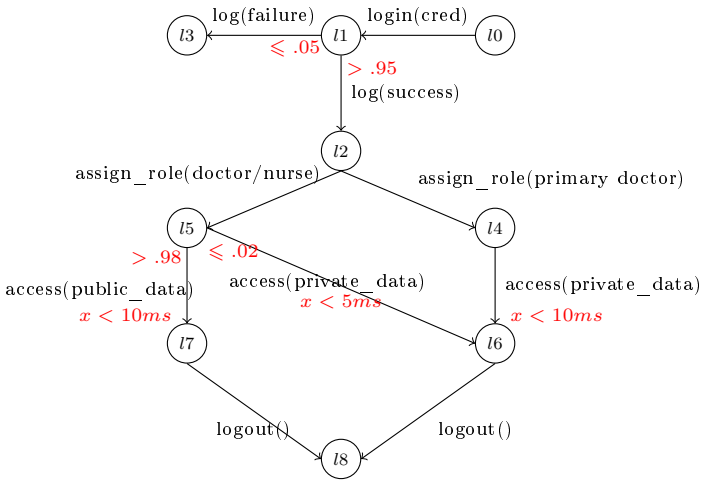


Figure 3: Probability constraints of the eHealth service

We use annotations on the system model in Definition 3.6 as a way to represent configuration constraints regulating the support of a given property. A cloud configuration constraint is a set of pairs $\Psi(prop, v)$, where *prop* represents a cloud configuration property and v its value.

We then define a labeling function $\lambda_c{:}E{\rightarrow}\Psi(prop, v)$ that associates a configuration constraint $\psi{\in}\Psi(prop, v)$ with a transition $(l_i,l_j){\in}E$ of m_{prob}. Based on λ_c, we define a trustworthy system model m_r as an extension of the system model m_{prob} in Definition 3.6.

Definition 3.7 (Trustworthy system model (m_r)). *Model m_r is a 8-tuple $(L, l_0, A, E, F, \lambda_t, \lambda_{prob}, \lambda_c)$, where L, A, E, l_0, F, λ_t, and λ_{prob} are defined in Definition 3.6, and λ_c assigns a label $\lambda_c((l_i,l_j))$, corresponding to the configuration constraint defined for each transition (l_i,l_j).*

We note that in case a transition $(l_i,l_j){\notin}E$, $\lambda_t((l_i,l_j))$ refers to the path(s) connecting l_i and l_j.

Appendix D illustrates the pseudocode of our algorithm for time constraint verification (CCV). It takes as input a trustworthy system model m_r and produces as output either *success* (1), if each configuration constraint in the corresponding path is satisfied, or *failure* (0), otherwise, with the list of transitions violating configuration constraints.

Failures in the verification of configuration constraints mean that one or more configuration requirements are not met. Failures at this stage can have serious implications for cloud providers promising configurations that are not present, or in case of misconfigurations/attacks. Thus, if a configuration constraint is not satisfied the system needs to be tested to check the problem and take the appropriate actions to solve it.

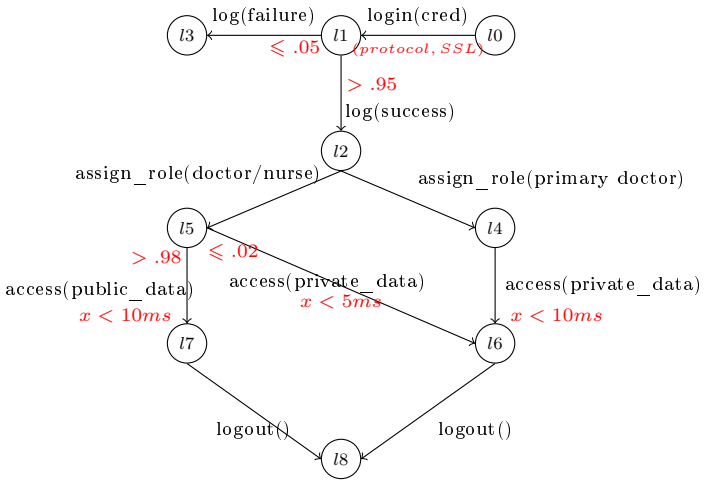


Figure 4: Configuration constraints of the eHealth service

Example 3.4. *We extend Example 3.3 by adding one configuration constraint stating that the communications during the login phase need to be protected by security protocol SSL (Figure 4). This property needs to be continuously checked to verify the support of corresponding certificates.*

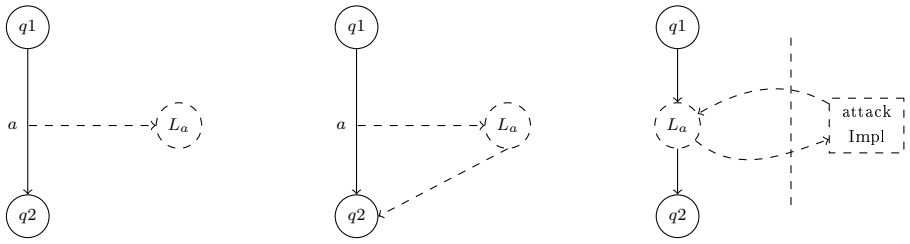
3.5. Attack paths

The modeling of service interactions, time, probability and configuration constraints goes beyond the simple checking of the correct system behavior. It also permits to indirectly verify the conformance between the running system and its security requirements. For example, when the distribution of invoking certain services undergoes a rapid change, it might be an indicator of a security breach. As another example, traces that do not reflect the annotations in the models might represent a possible basis for security threats.

Attack path verification permits to even strengthen security checks, by verifying robustness of the ToC against known attacks.[2] We define an attack model as a perturbation of the system model in Definition 3.1, based on well-known fault injection techniques, with the purpose of representing attack vectors.

Definition 3.8 (Attack Model m_{AT}). *Model $m_{AT}{=}(L', l_0, A', E', F)$ is an extension of $m{=}(L, l_0, A, E, F)$, where $L'{=}L{\cup}L_a$, $A'{=}A{\cup}A_a$, and $E'{=}E{\cup}E_r{\cup}E_w$, with L_a a finite set of attack states, A_a a finite set of attack actions, $E_r{:}L'{\times}A_a{\rightarrow}L_a$*

[2]We recall that we are not implementing an intrusion detection system; rather, our goal is to implement a trustworthy certification process, which is based on the continuous verification of the model representing the ToC and used to verify its properties in the production environment.



(a) Data Interception (b) Data Forwarding (c) Data modification

Figure 5: Attack classes

a transition annotated with a read action, and E_w:$L_a \times A_a \rightarrow L'$ *a transition annotated with a write action.*

We note that our attack model m_{AT} can also be seen as a set of attack paths, which are used to generate attack traces. Attack traces implement attacks to a real system and show possible security failures. Three attack classes have been considered in our approach (Anisetti et al. (2013b)), namely *i)* data interception, *ii)* data forwarding, and *iii)* data modification. States L_a and actions A_a are used to model the attack classes as shown in Figure 5. We note that, depending on different situations, additional attack classes can be added to the model. Formally, attack traces are defined as follows.

Definition 3.9 (Attack Trace AT_i). *An attack trace* AT_i *is a sequence* $\langle a_1,\ldots,a_n \rangle$ *of actions, where at least one* a_j *is part of the attack model* m_{AT}.

Appendix E illustrates the pseudocode of our algorithm for attack path verification (APV). It takes as input the system model, the attack model, and the corresponding attack traces, and produces as output either success (1), if all attack traces fail, or failure (0), otherwise, with the list of successful attack traces. The system is considered to be secure if none of the attack traces can follow the corresponding path in the system model in Definition 3.1 and retrieve successful results.

Example 3.5. *Let us consider model m in Figure 1. Functionality* login *is perturbed according to an attack model of class data interception (Figure 5(a)) modeling a replay attack. The man-in-the-middle attacker intercepts a call to the login service, retrieves the login data, and replay them to authenticate to the system. If the system grants access to the attacker, that is, the same result of an authorized users sending a call to the login service is retrieved, we can conclude that the login service is vulnerable to the replay attack.*

Our certification process and framework use the trustworthy system model in Definition 3.7 and the attack model in Definition 3.8 to verify system model correctness.

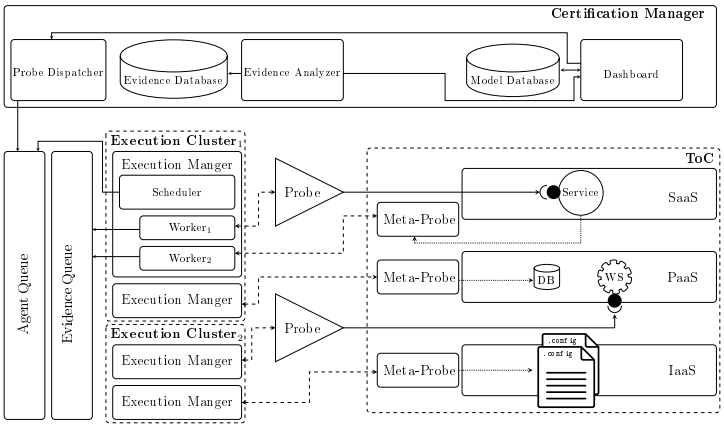


Figure 6: A simplified view of the framework architecture

4. Certification Framework

Our certification framework has the main responsibility of managing a trustworthy certification process and verifying the validity of corresponding certificates. Its architecture and components support any (semi-)automatic assurance process, including audit and compliance processes.

Figure 6 presents a view of our framework architecture. The framework is composed of a *certification manager* that orchestrates the whole certification and model verification processes, and a set of *execution clusters*, each managing and executing a set of *agents* collecting the evidence at the basis of certification and verification activities.

The certification manager stores all configurations and information needed to verify the correctness of the model (model structure, time constraints, probability constraints, configuration constraints, attack paths) and evaluates it once a sufficient amount of evidence is available. Certification manager is composed of the following modules:

- *Dashboard*: a web application providing an interface to manage certification and model verification processes.
- *Model Database*: a database storing all configurations and rules for certification operations and model verification.
- *Probe Dispatcher*: a module implementing a communication channel between certification manager and execution clusters to require agent execution.
- *Result Database*: a time-series database automatically storing all evidence coming from execution clusters at the basis of certification and verification activities.

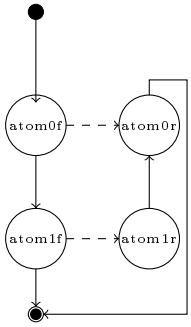
- *Evidence Analyzer*: a module analyzing the evidence in the result database and verifying the model correctness based on specified constraints.

Execution clusters are composed of execution managers that manage, deploy, and run agents on demand following requests from the certification manager on a full-duplex channel, where collected evidence are also exchanged. Communications are implemented using queues and the producer-consumer pattern. An execution cluster deploys multiple execution managers, each composed of the following modules:

- *Scheduler*: a module attached to the agent queue and constantly waiting for messages from the certification manager. Upon a request arrival, it dispatches the task to a worker.
- *Worker*: a module directly connecting to agents in order to send them all configurations needed for their execution. Upon agent execution is completed, the resulting evidence is sent to the certification manager through the evidence queue.

We note that our design based on multiple certification managers allows to scale the computational capacity of the system, since the certification manager can choose the most appropriate execution cluster to exercise the ToC. We also note that agents can be of two types working at different levels of granularity. The first agent type, called *probe*, includes *testing* and *monitoring* functionalities. They directly exercise the ToC by executing test cases and/or by monitoring events, to the aim of evaluating the behavior (including attack scenarios) of the system under certification. Their main goal is to identify possible inconsistencies between the evidence retrieved in a laboratory environment and the one retrieved in the production environment. The second agent type, called *meta-probe*, is used for the verification of contextual (i.e., time, probability, configuration) constraints and system model correctness. They indirectly exercise the ToC by observing real and synthetic execution flows (traces) of the system under certification. Their main goal is to identify violations of time, probability, and configuration constraints, and inconsistencies of the execution flows.

Probes and meta-probes have the same structure (see Figure 7). They are composed of different atom operations that accept as input configuration parameters and results produced by previous operations, and returns as output evidence on the behavior of the system. An example of probe structure is showed in Figure 7. A probe is modeled as a State Transition System (STS) with two main flows of the same size (Figure 7(a)): *i)* the forward chain and *ii)* the rollback chain. The forward chain contains all the states that should be executed if there are no exceptions during the probe execution, otherwise the flow is redirected on the corresponding rollback state and continues on the rollback chain. This approach is designed to guarantee that the ToC can be always restored to the initial state. The example in Figure 7 shows a probe with 4 states (2 forward states, 2 rollback states). The order of execution and association between forward and rollback states are specified by the configurations in method



(a)

```
class SimpleEmptyProbe(Probe):

    def atom0f(self, inputs):
        #reading input
        port=self.testinstances["config"]["port"]
        ...
        return True
    def atom0r (self, inputs):
        ...
        return False

    def atom1f(self, inputs):
        #return evidence
        self.result.put("evidence-key","evidence-value")
        ...
        return True
    def atom1r (self, inputs):
        ...
        return False

#Definition of execution order
    def appendAtomics(self):
        self.appendAtomic(self.atom0f, self.atom0r)
        self.appendAtomic(self.atom1f, self.atom1r)
```

(b)

Figure 7: Probe script in python. The probe is composed of two atom operations (*atom0*, *atom1*) with the corresponding rollback operations (*atom0r*, *atom1r*). Method *appendAtomics* specifies the order and matching of atom operations. *atom0* is executed first; *atom1* can access the results from *atom0* according to its definition.

appendAtomics (Figure 7(b)). The complete code of some example probes are described in Section 5.2 and available for interested readers at `https://github.com/SESARLab/Security-Constraint-Cloud-Service-Composition`.

In some cases, probes and meta-probes can be used interchangeably. The choice depends on the specific scenario, and balance the quality of the retrieved evidence and the required level of access to the cloud infrastructure. Meta-probes connect to interfaces (hooks) provided by the cloud provider with limited access to the cloud backend; probes directly access the cloud backend and can manage part of it. As an example, both probes and meta-probes can be used to verify property confidentiality by encryption of a cloud storage. Probes require direct read/write access to the storage, while meta-probes only check whether the storage is configured to encrypt data without any access to the stored data. The use of meta-probes comes with some non-negligible advantages. First, they do not require the cloud providers to open their system to the outside and release sensitive data. Second, they verify support of a property without interfering with the normal execution of the system. Third, they introduce lower overhead. These advantages come at the price of a reduced quality of the evidence. In the following, we denote as structure, time, probability, and configuration (meta-)probes, (meta-)probes dealing with system model structure, time constraints, probability constraints, and configuration constraints, respectively.

In Section 5, we present the working of our certification framework in a real scenario. To this aim, we implemented a certification manager that collects evidence on the behavior of the system (probes) and evidence on system correctness

(meta-probes), and uses them to manage a certification process and its certificate life cycle. We note that different assurance/adaptation frameworks (e.g., audit and compliance frameworks) can be implemented by simply changing the way in which evidence is used by the certification manager.

5. Experimental evaluation

We evaluate our trustworthy certification process in the context of the eHealth application provided by ATOS SA, a major IT player, deployed on top of Openstack. We first describe our scenario discussing the security requirements at both application and infrastructure layers (Section 5.1). We then present the setup and deployment of our certification framework, the implemented probes and meta-probes for security property certification, and the results of our evaluation (Section 5.2). We finally discuss the achievements of our certification framework and present some future work (Section 5.3).

5.1. Reference Scenario: ATOS SA eHealth Application

Our reference scenario is the eHealth application provided by ATOS SA, a telemedicine cloud application for patients suffering from dementia. ATOS SA is an international information technology services company, and one of the biggest system integration players in Europe. The eHealth application aims to support patients in their everyday life with a specific and dedicated tool, as well as caregivers and clinicians during the patient treatment. It is composed of three distributed components:

- *GUI application server* (GAS) is a self-contained server component that manages eHealth users, the data resulting from the monitoring of patients' biometric indicators, and tasks and questionnaires assigned to patients. It offers a web graphic interface to access all service features.
- *Engine application server* (EAS) is a self-contained SOAP-based server component that acts as an interface between the eHealth application components. It hosts the functionalities offered by the system through web services, namely: *i)* management of roles, permissions, connections, and transactions with *the database server*, *ii)* information provisioning to third party services and applications, *iii)* execution of the main business application logic (e.g., patients' data, warnings, and message processing).
- *Database server* (DS) is a MySQL DBMS. It stores and manages all persistent information, such as personal data of patients, caregivers, and doctors, patient's medical measurements, and questionnaire data with patients' responses, to name but a few. All data are manipulated via the *engine application server* ensuring data integrity and consistency.

eHealth application is designed to work on OpenStack infrastructure. OpenStack is an open source IaaS solution providing functionalities for the management and monitoring of infrastructure resources. It is becoming a standard

de facto due to its wide adoption by big IT companies and provides different services at the basis of eHealth deployment.

- *Identity service (Keystone)*: it provides authentication and authorization functionalities to all OpenStack services.
- *Compute service (Nova)*: it provides Virtual Machine (VM) management through abstraction layers that support different hypervisors.
- *Block storage (Cinder)*: it provides persistent storage. It supports a full life cycle management for block storage, and access control and encryption functionalities.
- *Network service (Neutron)*: it provides IP management, DNS, DHCP, load balancing, firewall policies, and VPN management.
- *Database service (Trove)*: it provides a database as a service for OpenStack. It is designed to run entirely on OpenStack, with the goal of allowing users to quickly and easily utilize the features of a relational or non-relational database, but without the burden of handling complex administrative tasks.

The eHealth application components are mapped on OpenStack as follows: *database server* is deployed using *Trove* to speed up the deployment and to continuously monitor the database cluster status; *engine application server* and *GUI application server* are deployed as VMs. These VMs are managed by *Nova*, while their storage is managed by *Cinder*. Figure 8 presents the architecture of the eHealth application, showing the deployment of its components on top of Openstack. For conciseness, when clear from the context, we will refer to the eHealth deployment in Figure 8 by the shortened name 'eHealth'. The application layer of eHealth is composed of the public interfaces exposed by *engine application server* and *GUI application server*, while the infrastructure layer is composed of the Openstack services including the *database server*.

5.1.1. Security aspects

The notion of cloud security depends on the service model and the requested level of assurance. We adopted the security domain concepts and classification collected by the OpenStack security group (*OSSG*) in the OpenStack Security Guide OpenStack Foundation (2016), and mapped them to our eHealth scenario. According to OSSG, "*a security domain comprises users, applications, servers or networks that share common trust requirements and expectations within a system*" (OpenStack Foundation (2016)). In our scenario, *users* are the Openstack cloud administrators, cloud tenants, and eHealth users, *applications* are the eHealth services deployed on Openstack VMs, *servers* are the OpenStack services, and *networks* include both internal and external networks. Security domains are classified as follows.

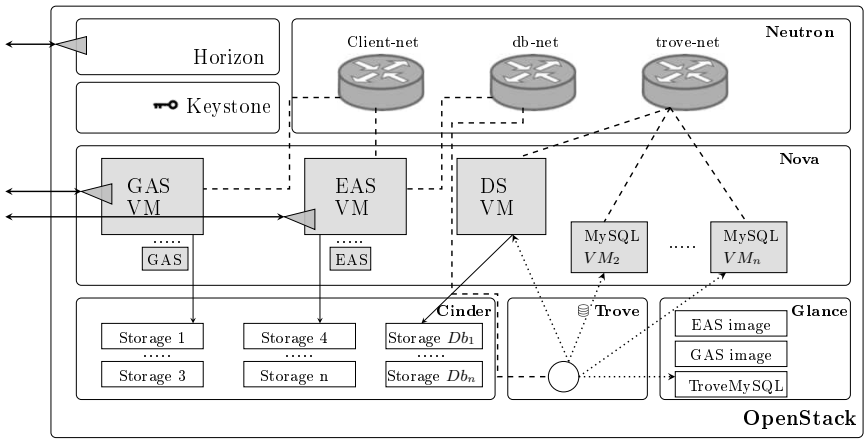


Figure 8: eHealth application on top of Openstack infrastructure. Solid triangles represents public interfaces

- *Public*: it refers to the untrusted portion of the cloud. Each data posing confidentiality or integrity issues should be protected by using external controls before transiting the public domain.
- *Guest*: it refers to instance-to-instance or application-to-application communications. It considers all data produced by cloud applications/VMs.
- *Management*: it refers to communications and data produced by tools and services part of the cloud business core.
- *Data*: it refers to information concerning the storage services.

The evaluation of our certification framework builds on the mapping between eHealth and security domains (Figure 9), and focuses on security properties derived from eHealth security concerns. When dealing with cloud applications, we must consider security concerns of both the application (eHealth) and the underlying cloud layers (OpenStack hosting eHealth). eHealth treating sensitive medical data has a lot of security and privacy concerns. All eHealth sensitive data need to be exchanged and stored in a confidential way. Any access to confidential data, either from service interfaces or directly through infrastructure ports, must be restricted via authorization rules preventing from unauthorized access, as well as identity theft. OpenStack supporting the management and storage of sensitive data has also several security concerns. For these concerns, we refer to the OpenStack Security Guide (OpenStack Foundation (2016)) including security advices/considerations and keeping track of vulnerabilities, which are released as security notes.

5.2. Certification of eHealth

We deployed our framework and setup our certification process for the verification of a set of security properties for eHealth application on a 32 physical

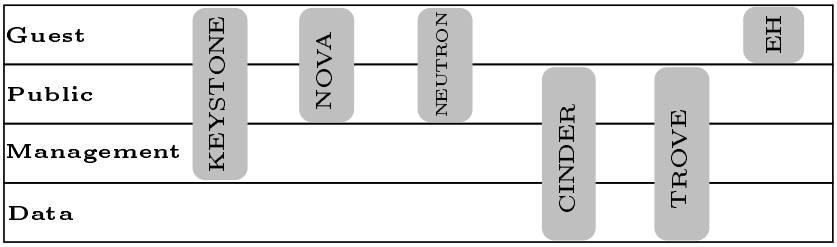


Figure 9: Mapping between OSGG security domains and eHealth. eHealth application layer is denoted as *EH*, eHealth infrastructure layer is represented by Openstack services.

machines ProLiant BL2x220c G6 equipped with 12 Intel Xeon X5660 cores at 2.80 GHz, 24GB RAM, 120GB SATA HDD. The storage node is composed of 1TB offered as Network File System (NFS). Our machine hosts a replica of the eHealth production cloud, where OpenStack has been installed with *i)* no redundancy, *ii)* Trove, Sahara, and Heat services in addition to core services, and *iii)* TLS/SSL module enabled for all services. Security properties have been verified according to relevant security aspects in the OpenStack security guide (Section 5.1.1) and requirements imposed by eHealth.

In the following, we provide a section for each model verification step in Section 3 presenting certification activities, describing the code of our (meta-)probes, and discussing the retrieved results. In particular, for time (Section 5.2.2), probability (Section 5.2.3), configuration (Section 5.2.4) constraints, and attack paths (Section 5.2.5), we present the certification activities executed on the eHealth application describing: *i)* the goal, *ii)* the scenario, *iii)* the target of certification in terms of eHealth components, *iv)* the implementation of probes and meta-probes (including their code), and *v)* an analysis of the retrieved results. For model structure (Section 5.2.1), instead, we provide an evaluation based on simulation, which goes beyond an evaluation based on a single reference scenario with a single system model. Model structure verification, which is a mandatory step of our verification process and is at the basis of the others verification steps, requires a more extensive evaluation implemented using model perturbation.

5.2.1. Model Structure Verification

We show how our approach can detect model inconsistencies and changes in the running system can be reflected in the model. To this aim, we developed a prototype composed of two main modules, namely, *consistency checker* and *model adapter*, and a model generator that generates random models and provides the ability to perturb them at various degrees (e.g., add new paths, remove existing paths).[3] The *Consistency checker* receives as input a service

[3]The model generator code can be found in `https://goo.gl/4fQRhE`.

model and a set of real and synthetic traces,[4] and returns as output inconsistent traces with the corresponding type of inconsistency. *Model adapter* receives as input the results of the consistency checker and, according to them, generates as output a refined model.

A data set has been generated to test the effectiveness of our prototype as follows. A model m has been defined manually to represent the correct implementation of a portion of the eHealth system. The model m is composed of 20 different paths. Using the model generator, we randomly generated 1000 inconsistent models by adding random inconsistencies to m. Inconsistent models are such that 10%, 20%, 30%, 40%, or 50% of the paths are different (e.g., missing, new) from the paths in m. To simulate realistic customer behaviors, we extended m by adding a probability P_i of executing each path $p_i \in m$ such that $\sum_i P_i = 1$. Probabilities are taken randomly from a normal probability distribution such that there exist few paths whose probability of being invoked tends to 0. Real and synthetic traces are produced using m extended with probabilities.

First, using the *consistency checker*, we verified inconsistent models in the data set and retrieved inconsistent traces together with the type of inconsistency (using the algorithm MSV in Appendix A)). Then, using the *model adapter*, we built a refined model of each inconsistent model, and evaluated how much these models approximate the correct model m. The results show that the refined models covered 64% of paths in m on average, with an increase of 28% on the average coverage of the inconsistent models. Also, 17% of the inconsistent models were able to cover the entire model m (i.e., 100% coverage), while 0% of the inconsistent models in the data set covered the entire model by construction. Figure 10 shows a Box and Whisker chart presenting more detailed results on the basis of the rates of differences (i.e., 10%, 20%, 30%, 40%, 50%) introduced in the inconsistent models. Figure 10 shows that an increased perturbation level results in a decreased ability to achieve a full coverage of the initial model. The median value decreases in such a way that the complete coverage is achieved in the last category (50%) as an outlier. Nevertheless, for all perturbation levels, 50% of the models recovers at least 40% of the inconsistencies. Additionally, in the first case (10% perturbation), more than 50% of the models were completely recovered; therefore, both the median and the maximum coverage are equal to 1. These results are mainly due to the fact that paths at low probability are invoked with low probability. If these paths are already specified in an inconsistent model, then a synthetic trace can be generated to evaluate them (if no real traces are observed) and the refined model covers 100% of m. Otherwise, they remain *hidden* impairing the ability of model adapter to produce a refined model that covers 100% of m.

5.2.2. Time Constraints - "Access Performance"

The access performance (responsiveness) of an eHealth service is crucial to save life. A prompt access to patience information in emergency cases is even

[4] We remark that synthetic traces are generated by ad hoc testing.

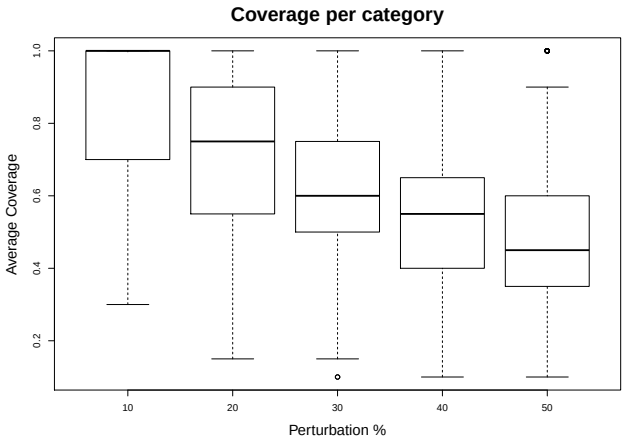


Figure 10: Refined model coverage of m.

more crucial. This property has an impact both at service and infrastructure layer measuring the total time needed to retrieve patience information upon request (response time).

Goal. The goal of the certification process is to check whether the response time of function *access(private_ data)* is under a given threshold m_i (m_i=500ms in our example). Time is measured from outside the application as a nurse or a doctor interacts with the application.

Scenario. A user logs into the eHealth application during an emergency situation and requires private data of a patience not under her control.

ToC. It consists of the eHealth application with particular focus on function *access(private_ data)*. It insists on *guest* and *data* domains in Figure 9. In fact, to access data, the system uses the whole architecture: the request passes through the EAS, accesses the DS, and has an impact on the whole infrastructure, using storage, network and computational power.

Implementation. The evaluation is carried out using a time probe continuously requiring AES to execute function *access(private_ data)*. The probe calculates the response time and, in case it is higher than m_i, it notifies back the certification manager. We note that each response time is evaluated independently, since every call must guarantee the requested access performance. Figure 11 shows the code of our probe, which is available at `https://goo.gl/zXP8PN`. The probe receives as input *i)* login credentials, *ii)* patience id, *iii)* time threshold m_i. It then calculates and returns as output the response time (lines 15-17).

Analysis of results. Property access performance is certified iff the time probe never returns a negative evaluation. We run the probe for 2 days and, since the response time was always under the threshold m_i, the property was certified.

```
from requests.auth import HTTPBasicAuth
from requests import request
import time
from driver import Driver

class RensponseTime(Driver):
    def login(self,inputs=None):
        #preprare Header for http basic authentication
        login_ti=self.testinstances.get("login", None)
        auth = HTTPBasicAuth(login_ti.get('user'), login_ti.get('password'))
        return auth
    def get_response(self, inputs=None):
        time_m=self.testinstances.get("call").get("time threshold")
        patience_id=self.testinstances.get("call").get("patience id")
        auth=inputs
        start_time=time.time()
        request.get('https://aes.ehealth.local/access-private-data/'+patience_id, auth=auth)
        response_time = time.time() - start_time
        if response_time <= time_m:
            return True
        else:
            return False
    def logout(self, inputs):
        return False
    def rollback(self, inputs):
        self.result.put_value("exception", "The probe exit because an exception")
        return False
    def appendAtomics(self):
        self.appendAtomic(self.login, self.logout)
        self.appendAtomic(self.get_response,self.rollback)
```

Figure 11: Code of the time probe for evaluating property access performance

5.2.3. Probability Constraints - "Authorization-Based Privacy"

Authorization-based privacy is crucial for eHealth, where the privacy of patients' data is paramount. More specifically, authorization-based privacy requires that only authorized personnel can access patients' data. An exception to this rule applies for emergency situations (*break the glass* (BG) scenario in Example 3.1), where all doctors and nurses can temporary access patients' data. This property has an impact at application layer where eHealth interfaces are provided and BG scenario is modeled.

Goal. The goal of the certification process is to check that every access to eHealth satisfies the defined RBAC (role-based access control) policies. Also, it aims to verify that the BG scenario is not abused by checking probability constraints.

Scenario. eHealth users (e.g., nurses, doctors) request access to resources for which they are authorized (possibly in a BG scenario). Access requests are collected in a log for a predefined window of time.

ToC. It consists of the eHealth application and insists on *guest* domain in

```
[22/Aug/2017 14:06:28] user_1 login(cred) 200
[22/Aug/2017 14:06:54] user_1 log(failure) 200
[22/Aug/2017 14:06:58] user_2 login(cred) 200
[22/Aug/2017 14:07:24] user_3 login(cred) 200
[22/Aug/2017 14:07:28] user_2 log(success) 200
[22/Aug/2017 14:07:54] user_3 log(success) 200
[22/Aug/2017 14:07:54] user_2 assign_role(doctor/nurse) 200
[22/Aug/2017 14:07:58] user_2 access(public_data) 200
[22/Aug/2017 14:08:24] user_2 logout() 200
[22/Aug/2017 14:08:28] user_4 login(cred) 200
[22/Aug/2017 14:08:30] user_4 log(success) 200
[22/Aug/2017 14:08:54] user_3 assign_role(doctor/nurse) 200
...
```

Figure 12: Log file (excerpt)

Figure 9.[5] It mainly focuses on the flow described in Example 3.3.

Implementation. The evaluation is carried out by *i)* a monitoring probe controlling the correctness of every access to eHealth and *ii)* a probability meta-probe checking abuses such as in cases of BG scenarios. The monitoring probe is implemented using different monitors working at application layer, which observe all accesses and evaluate them against RBAC policies. The probability meta-probe evaluates the frequency of accesses at application layer to uncover higher probability of utilization with respect to the one designed by the certification authority (see for instance BG scenario in Example 3.3 and Figure 3).

For brevity, we provide details on the probability meta-probe only. The probability meta-probe periodically accesses the application logs collected by the monitoring probe using an SSH service and filter them using the selected time window. An example of log file is presented in Figure 12 (the entire log file is available at `https://goo.gl/xntHhQ`) and follows the pattern *[<time>] <user> <operation> <code>*.[6]

Figure 13 shows the code of our meta-probe, which is available at `https://goo.gl/a3JHjx`. The meta-probe receives as input *i)* SSH credentials, *ii)*log file path, *iii)* operation sequences with expected probabilities. It then analyzes each entry in the log and maps them over the flow sequences in Figure 3. Once all flows are mapped, the meta-probe calculates frequencies and returns them as output. The meta-probe starts with the *ssh_connection* (lines 6-21), which reads the SSH credentials and connects it to the host. Then, it reads the log file with *log_connection* (lines 22-30) and analyzes it line-by-line in *log_analysis* (lines 31-47).

Analysis of results. Property *authorization-based privacy* is certified iff both the probe and the meta-probe return a positive evaluation. We run them on a set of 700 operations distributed in 138 requests collected in a log. Since each access was successfully evaluated against RBAC policies and the retrieved

[5] We note that similar checks can also be applied at infrastructure.

[6] We note that, for privacy reasons, the logs used in this evaluation are synthetic.

sequence	expected frequency	number of flows found	frequency
login(cred), log(failure)	0.05	2	0.014
login(cred), log(success),*	0.95	136	0.986
login(cred), log(success), assign_role(doctor/nurse), access(public_data), *	0.98	98	0.98
login(cred), log(success), assign_role(doctor/nurse), access(private_data), *	0.02	2	0.02

Table 1: Meta-probe results

frequency satisfied the probabilities in the model (see Table 1)

5.2.4. Configuration Constraints -"Storage Confidentiality"

Storage confidentiality is a crucial property for systems managing sensitive personal data. In our scenario, it impacts infrastructure layer only since eHealth relies on the storage service provided by Openstack, where data are stored enabling cryptographic encryption at file system level.

Goal. The goal of the certification process is to check if stored data are managed in a secure way, making eHealth robust against unauthorized access to sensitive information. To address this requirement eHealth secured the virtual storage by using infrastructure-layer encryption mechanisms.

Scenario. The eHealth application requires an encrypted storage to store its information before being attached to VMs (e.g., at deployment time, while scaling, or for load balancing). *Cinder* manages the persistent storage, and *Nova* provides an encryption mechanism and features to attach it to VMs. *Cinder* and *Nova* use an encryption function based on Dm-Crypt (Benjamin et al. (2013)), which prevents unauthorized access to sensitive information creating a Linux Unified Key Setup (LUKS) storage.

ToC. It includes *Nova* and *Cinder* services, and insists on security domains *data* and *management* in Figure 9.

Implementation. The evaluation is carried out using a configuration meta-probe connecting to *Cinder* service to check that every storage is correctly set as encrypted. Figure 14 shows the code of our meta-probe, which is available at `https://goo.gl/1pkim8`. The probe receives as input *i)* OpenStack credentials, *ii)* volume list and the encryption type for the given volume. The probe connects to OpenStack APIs (lines 7-17) and checks that every requested volume is encrypted (lines 18-33).

Analysis of results. Property *storage confidentiality* is certified *iff* the configuration meta-probe successfully verifies that all available storage is configured as

```
from userManager import users,us,build_users
from ssh_connector import SSHClient
from driver import Driver

class FrequencyProbe(Driver,SSHClient):
    def ssh_connection(self,inputs=None):
        ssh_connection_ti =
            self.testinstances.get("connect_to_server", None)
        assert not ssh_connection_ti is None
        hostname = ssh_connection_ti.get("hostname")
        port = ssh_connection_ti.get("port")
        username = ssh_connection_ti.get("username")
        password = ssh_connection_ti.get("password", None)
        assert not password is None
        self.ssh_connection = self.ssh_connect(
            hostname=hostname,
            username=username,
            port=port,
            password=password
        )
        return True
    def log_connection(self, inputs=None):
        #connect to log host and read log file
        logs_ti = self.testinstances.get("logs", None)
        _stdin, _stdout, _stderr =
            self.ssh_connection.exec_command(
                "cat "+logs_ti.get("log_path")
                )
        out = _stdout.readlines()
        return out
    def log_analysis(self,inputs=None):
        #parse model input
        model_ti =self.testinstances.get("logs", None)
        model=[]
        for item in model_ti:
            model.append(item)
        #parse log file
        myfile=inputs.split("\n")
        build_users(myfile)
        analyselog(model)
        frequency=[0]*model[len(model)-1]["flow"]
        for m in model:
            frequency[m["flow"]]+=m["count"]
        for m in model:
            if m["p"] > (m["count"]/frequency[m["flow"]]):
                result=False
        return result
    def close_ssh_connection(self, inputs=False):
        try:
            self.ssh_connection.close()
        except:
            pass
        return inputs
    def rollback(self, inputs):
        self.result.put_value("exception",
                "The probe exit because an exception")
        return False
    def appendAtomics(self):
        self.appendAtomic(self.ssh_connect, self.close_ssh_connection)
        self.appendAtomic(self.log_connection,self.rollback)
        self.appendAtomic(self.log_analysis,self.rollback)
        self.appendAtomic(self.close_ssh_connection,self.rollback)
```

Figure 13: Code of the probe for evaluating property authorization-based privacy

```
from keystoneauth1 import loading
from keystoneauth1 import session
from cinderclient import client
from driver import Driver

class ProbeStorageCinder(Driver):
    def authentication(self,inputs=None):
        loader = loading.get_plugin_loader('password')
        auth = loader.load_from_options(
            auth_url=self.testinstances["openstack credential"]["
    OS_AUTH_URL"],
            username=self.testinstances["openstack credential"]["
    OS_USERNAME"],
            password=self.testinstances["openstack credential"]["
    OS_PASSWORD"],
            project_id=self.testinstances["openstack credential"]["
    OS_PROJECT_ID"],
            user_domain_name=self.testinstances["openstack credential
    "]["OS_USER_DOMAIN_NAME"]
        )
        sess = session.Session(auth=auth)
        return sess
    def check_volumes(self, inputs=None):
        encrypted_type=self.testinstances.get("volumes").get("encrypted
     type")
        volumes_ti=self.testinstances.get("volumes").get("list")
        volumes=volumes_ti.split(',')
        sess=inputs
        cinder = client.Client('2', session=sess)
        volume_list = cinder.volumes.list()
        found=len(volumes)
        for v in volume_list:
            if v.name in volumes:
                found=found-1
                if v.types != encrypted_type:
                    return False
        if found != 0
            return False
        return True
    def rollback(self, inputs):
        self.result.put_value("exception", "The probe exit because an
    exception")
        return False
    def appendAtomics(self):
        self.appendAtomic(self.authentication, self.rollback)
        self.appendAtomic(self.check_volumes,self.rollback)
```

Figure 14: Code of the probe for evaluating property storage confidentiality

Linux Unified Key Setup (LUKS) storage at infrastructure layer. Since eHealth used an encrypted storage, the meta-probe returned a positive result and the property was certified.

5.2.5. Attack Paths - "Replay Attack Vulnerability"

eHealth protects access to its services by means of a login service implementing a password-based authentication. Replay attack is a possible way for an attacker to bypass the login service and obtain unauthorized access to the system. A replay attack assumes a man in the middle with the possibility to capture, store, and replay traffic exchanged between a client and the server.

Goal. The goal of the certification process is to check if the login service of the eHealth application is vulnerable to a replay attack.

Scenario. An attacker captures, stores, and replays a call to the login service sent by an eHealth user.

ToC. It includes the network infrastructure and the login service of the *eHealth* application, and insists on security domains *Public* and *Guest* in Figure 9.

Implementation. The evaluation is carried out using a probe with the requested attacker capabilities. Figure 15 shows the code of our probe, which is available at `https://goo.gl/psrJn3`. The probe first checks if a replay attack is possible, that is, it checks if the communication channel is encrypted with TLS or not using NMAP (lines 11-12).[7] Then, if the channel is not encrypted with TLS, the probe captures a call to the login service and replay it on the eHealth login API (lines 13-26). If the login service returns a positive response, the attack is successful. The probe uses *tshark* to capture the traffic (`https://www.wireshark.org/docs/man-pages/tshark.html`) and *tcpreplay* to replay it (`http://tcpreplay.appneta.com/`).

Analysis of results. Property *replay attack vulnerability* is certified *iff* the probe successfully verifies that eHealth is not vulnerable against a replay attack. Since eHealth used a TLS encrypted channel with a strong encryption key (RSA-2048), the probe returned a positive evaluation and the property was certified.

5.3. Discussion

We showed how our trustworthy certification process can be applied to an industrial eHealth application deployed on top of OpenStack. Our evaluation presented the certification of different properties by means of both probes and meta-probes, available for download at `https://goo.gl/8iNmV4`.[8] Compared with our previous framework (Anisetti et al. (2015)), the current framework provides the following advantages. First, it supports verification of *i)* indirect constraints (i.e., time and probability) that would affect a certification process (e.g., property *authorization-based privacy*), *ii)* properties by simply checking the configuration of the underlying cloud layers (e.g., property *storage confidentiality*), *iii)* attack paths potentially invalidating security properties. As an example, in case *i)*, we checked potential abuses of the break-the-glass scenario, which would invalidate a certification process; in case *ii)*, we reached the same results of the testing approach in Anisetti et al. (2015) (adding and deleting volumes in production), requiring less resources (only a check of storage configuration); in case *iii)*, we checked robustness against replay attacks, which would

[7] RFC 5246 (`https://tools.ietf.org/html/rfc5246#appendix-F.2`) states that the use of TLS prevents from replay attacks.

[8] We note that our evaluation does not prove absolute support for the property, while it can profitably verify that some (security) features/mechanisms are in place and work correctly. Whether this is sufficient or not for releasing a certificate is under the bailiwick of the certification authority signing the certificate.

```
import subprocess
from requests.auth import HTTPBasicAuth
from requests import request
from libnmap.process import NmapProcess
from libnmap.parser import NmapParser, NmapParserException
from logger import check_change_value
from nmap import scan_nmap
from driver import Driver

class ReplayAttack(Driver):
    def nmapRun (self, inputs):
     return scan_nmap(self.testinstances["config"]["host"],self.
    testinstances["config"]["port"])
    def tcpdump(self,inputs):
      eth=self.testinstances["dump"]["interface"]
      if input == True:
        return True
      else:
        bash_command = "tshark -i " + eth + " -w test.pcap -F libpcap -
    a duration:200"
        process = subprocess.Popen(bash_command.split(), stdout=
    subprocess.PIPE)
        login_ti = self.testinstances.get("login", None)
        auth = HTTPBasicAuth(login_ti.get('user'), login_ti.get('
    password'))
        r=request.get('https://aes.ehealth.local/login', auth=auth)
        result_dict=r.json()
        value=result_dict["token"]
        output, error = process.communicate()
        return value
    def tcpreplay(self,inputs):
      if input == True:
        return True
      else:
        bash_command = "tcpreplay --intf1=en0 test.pcap"
        process = subprocess.Popen(bash_command.split(), stdout=
    subprocess.PIPE)
        output, error = process.communicate()
        return check_change_value(output)
    def rollback (self, inputs):
      return False
    def appendAtomics(self):
        self.appendAtomic(self.nmapRun,self.rollback)
        self.appendAtomic(self.tcpdump,self.rollback)
        self.appendAtomic(self.tcpreplay,self.rollback)
```

Figure 15: Code of the probe for evaluating property “replay attack vulnerability"

invalidate a certification process for property authorization-based privacy. Second, the current framework supports verification of system model structure with respect to real execution traces, being able to discover discrepancies between the system behavior in laboratory and production environments (e.g., a backdoor in the case of property *authentication data confidentiality*). All these activities require deep inspection of the system and can be carried out using either probes or meta-probes. Meta-probes, compared to normal probes, are less invasive, do not require the cloud providers to open their system to the outside releasing sensitive data, and do not interfere with the normal execution of the system, at the price of a reduced loss of evidence quality. In addition, meta-probes do not enlarge the attack surface by introducing new testing/monitoring components

or interface. We note that probes are anyway fundamental to test/monitor the system in production by re-evaluating activities carried out in laboratory. Finally, as shown by the variety of verified properties, our framework can be easily applied to any classes of properties. Each class requires a reconfiguration of probe/meta-probes on the basis of the corresponding system/evidence collection model, leaving their working unchanged, as well as the working of the certification manager.

To conclude, we recall that our approach cannot guarantee that the model is always 100% valid. There are cases where the model is not reflecting the system implementation due to improper modeling and missing events from the execution traces (*hidden paths*). Hidden paths often refer to unlikely or unknown events that are difficult to catch if not properly modeled, such as undisclosed system vulnerabilities. Still, our approach guarantees that once a hidden path is executed, it is immediately caught (new path discovery in Section 3.1) and added to the model. To fill in this gap, in our future work, we plan to use fuzzing and mutation techniques to produce synthetic models used to reveal hidden paths.

6. Related Work

The rising interest in certification schemes for the cloud has fostered the definition of several certification processes focusing on different aspects of cloud services (e.g., security, performance). Examples of certification initiatives include Cloud Security Alliance STAR, EuroCloud, FedRAMP, TRUSTed Cloud Data Privacy Certification, to name but a few. Although cloud certification is reaching its maturity, it still lacks of a well-established certification process. Also, according to Lins et al. (2016a,c); Anisetti et al. (2014a), one of the major barriers that impedes cloud certification adoption is the lack of flexibility and the consequent difficulties in supporting cloud peculiarities, such as continuous architectural and environmental changes.

Recently, research on certification schemes for cloud services mostly considered testing-based and/or monitoring-based evidence (e.g., Anisetti et al. (2014a); Ardagna et al. (2016); Spanoudakis et al. (2012); Cloud Security Alliance (2017); Modic et al. (2016); Stephanow et al. (2016); Kunz and Stephanow (2017); Li et al. (2017); Stephanow and Khajehmoogahi (2017)). The testing and monitoring activities take place by directly interacting with the system under certification and checking specific properties, using a system model that is assumed to be a correct representation of the system itself. In this context, several approaches to cloud certification have been provided and are summarized in the following. Lins et al. (2016b) propose a conceptual architecture for cloud service certification based on continuous monitoring. The architecture relies on a loop that involves cloud service auditors and service providers: the auditors continuously monitor the services and report any inconsistencies to the providers, which are required to solve them. To increase cloud transparency and trustworthiness, the architecture assumes the involvement of cloud service customers who

are constantly updated about the services' health. Anisetti et al. (2014a) propose a chain of trust for cloud certification, which relies on model-based testing and monitoring techniques to collect certification evidence. Spanoudakis et al. (2012) present a hybrid, incremental, and multilayer framework for cloud certification based on service monitoring. Munoz and Mãna (2013) propose a different strategy based on trusted computing platforms for certifying cloud-based systems. Modic et al. (2016) present an approach, called Moving Intervals Process, for real-time cloud security assessment, which supports over-provisioning to assure better global security of the acquired cloud services. Stephanow et al. (2016) describe a test-based certification framework, based on randomized and non-invasive testing, for evaluating opportunistic providers. Di Cerbo et al. (2013) present an extension to the Digital Security Certificate in Kaluvuri et al. (2013) and an approach to continuous monitoring of certified properties. Bousquet et al. (2015) propose an approach that first uses a context-based language to express security and assurance properties on distributed resources, and then enforces these properties by automatically configuring available resources. Different research projects also focused on assurance and certification. SPECS project (Rak et al. (2013); Casola et al. (2016); Luna et al. (2017)) provides a security-oriented framework for cloud security SLA specification and life cycle management. The framework supports the definition of security contracts specifying the security guarantees offered by a cloud service, and including obligations and responsibilities of each involved party. It also monitors contract conformance at runtime and can re-negotiate them in case of incidents. In this context, security Service Level Agreement (secSLA) in Trapero et al. (2017) is an example of framework that integrates different security controls and is implemented as part of SPECS project. The approach in this paper could be applied within SPECS project, using defined SLAs as certificates associated with services. OPTET project (2013) aims to understand the trust relation between different stakeholders, offering methodologies, tools, and models that provide evidence-based trustworthiness. The project puts high emphasis on evidence collection during system development, enriched by monitoring and system adaptation to maintain its trustworthiness. Differently from the above works, the approach in this paper follows a systematic process in defining the system models and the evidence collection, and provides a trustworthy certification process that combines model-based certification of cloud services and verification of model correctness according to time, probability, and configuration constraints, and attack paths. Our approach continuously checks the consistency between the cloud service implementation and the service execution traces, accomplishing cloud peculiarities and providing continuous certification evaluation across cloud environmental changes. Although our examples focused on security, the approach is generic and can be applied to all quality properties that can be modeled using the presented formalism.

Model-based approaches have also been used to provide the evidence needed to certify cloud service quality. Model-based verification of software components is in fact increasingly becoming the first choice technique to assess the quality of (cloud) software systems, providing a systematic approach with solid

theoretical background. Existing studies consider different modeling techniques targeting specific aspects of cloud services (e.g., security, performance). Among them, several studies focus on modeling the system flow. Accorsi et al. (2011) propose Comcert, a Petri Net-based automated approach for the certification of business process compliance. Yeung (2006) defines a mapping from WS-CDL and BPEL4WS into communicating sequential processes, to verify whether the obtained orchestrations behave as expected in the corresponding choreography. Diaz et al. (2006) model temporal properties as a timed automata, then use UPPAAL model checker to simulate and analyze the system behavior. Maalej et al. (2013) present service compositions load testing based on Timed Automata to model the service workload. Abbors et al. (2013) present an approach and a tool (MBPeT) for services' performance testing, using Probabilistic Timed Automata to describe how users interact with services. Our paper improves on the above papers by providing an approach where the correctness of the system model is not given for granted. A methodology and algorithms for the verification of model correctness is provided and is at the basis of a trustworthy evidence collection and certification process.

More recently, the works in Anisetti et al. (2014a, 2015, 2014b); Ardagna et al. (2016) have provided the basic building blocks for the definition of a certification process addressing cloud peculiarities. In this paper, we consider advanced aspects aimed to guarantee the trustworthiness of the certification process in a dynamic cloud, including constraints on time relations between different states of the model, transition probabilities, and configurations, and attack paths for certificate validity.

7. Conclusions

The definition of trustworthy assurance techniques is the next step for strengthening the cloud position as the first service provider, also in those critical environments with strong non-functional (e.g., security, privacy) requirements. In the last 30 years, several assurance solutions, including audit, certification, and compliance, have been provided with different levels of trustworthiness and focusing on different systems (i.e., traditional software, service-based, and cloud-based systems). In this paper, we proposed a model-based trustworthy certification process for the cloud that accomplishes cloud requirements. Our approach starts from the assumption that no trustworthy certification is possible without a proper verification of model correctness, once the system is deployed in the production environment. We therefore provided a methodology (and corresponding algorithms) for the verification of model correctness against real and synthetic service execution traces, according to time, probability, and configuration constraints, and attack paths. Our methodology, while focusing on certification, is general and can be applied to any model-based assurance techniques.

Acknowledgments

This work was partly supported by the program "piano sostegno alla ricerca 2015-17" funded by Università degli Studi di Milano. We would like to thank Rodrigo Diaz and Maria Rosa Vieira Alvarez from ATOS SA for supporting the evaluation of our approach based on ATOS eHealth application.

References

Abbors, F., Ahmad, T., Truscan, D., Porres, I., April 2013. Model-based performance testing in the cloud using the mbpet tool. In: Proc. of ICPE 2013. Prague, Czech Republic.

Accorsi, R., Lowis, D.-I. L., Sato, Y., 2011. Automated certification for compliant cloud-based business processes. BISE 3 (3), 145–154.

Alford, T., Morton, G., 2009. The economics of cloud computing. Booz Allen Hamilton.

Anisetti, M., Ardagna, C., Damiani, E., September 2014a. A certification-based trust model for autonomic cloud computing systems. In: Proc. of IEEE ICCAC 2014. London, UK.

Anisetti, M., Ardagna, C., Damiani, E., Gaudenzi, F., Veca, R., June-July 2015. Toward security and performance certification of openstack. In: Proc. of IEEE CLOUD 2015. New York, NY, USA.

Anisetti, M., Ardagna, C., Damiani, E., Saonara, F., May 2013a. A test-based security certification scheme for web services. ACM TWEB 7 (2), 1–41.

Anisetti, M., Ardagna, C., Damiani, E., Saonara, F., May 2013b. A test-based security certification scheme for web services. ACM TWEB 7 (2), 1–41.

Anisetti, M., Ardagna, C. A., Damiani, E., El Ioini, N., November 2014b. Trustworthy cloud certification: A model-based approach. In: Proc. of SIMPDA 2014. Milan, Italy.

Ardagna, C., Asal, R., Damiani, E., Ioini, N. E., Pahl, C., Dimitrakos, T., June–July 2016. A certification technique for cloud security adaptation. In: Proc. of SCC 2016. San Francisco, CA, USA.

Ardagna, C. A., Damiani, E., October 2014. Network and storage latency attacks to online trading protocols in the cloud. In: Proc. of C&TC 2014. Amantea, Italy.

Bate, I., Bernat, G., Puschner, P., April-May 2002. Java virtual-machine support for portable worst-case execution-time analysis. In: Proc. of IEEE ISORC 2002. Washington, DC, USA.

Bellandi, V., Cimato, S., Damiani, E., Gianini, G., Zilli, A., 2015. Toward economic-aware risk assessment on the cloud. IEEE Security & Privacy 13 (6), 30–37.

Benjamin, B., Coffman, J., Glendenning, L., Reller, N., February 2013. VolumeEncryption. `https://wiki.openstack.org/wiki/VolumeEncryption`.

Bertholon, B., Varrette, S., Bouvry, P., July 2011. Certicloud: A novel TPM-based approach to ensure cloud IaaS security. In: Proc. of IEEE CLOUD 2011. Washington, DC, USA.

Bousquet, A., Briffaut, J., Caron, E., María Dominguez, E., Franco, J., Lefray, A., López, O., Ros, S., Rouzaud-Cornabas, J., Toinard, C., Uriarte, M., December 2015. Enforcing Security and Assurance Properties in Cloud Environment. In: Proc. of UCC 2015. Limassol, Cyprus.

Casola, V., De Benedictis, A., Rak, M., Modic, J., Erascu, M., 2016. Automatically enforcing security slas in the cloud. IEEE Transactions on Services Computing.

Chaudhuri, A., von Solms, S., Chaudhuri, D., 2011. Auditing security risks in virtual it systems. ISACA Journal 1 (16), `https://goo.gl/Fwvi3t`.

Cloud Security Alliance, 2017. CSA Security, Trust & Assurance Registry (STAR). `https://cloudsecurityalliance.org/star/`, Accessed in Date February 2015.

Di Cerbo, F., Bisson, P., Hartman, A., Keller, S., Meland, P., Moffie, M., Mohammadi, N., Paulus, S., Short, S., 2013. Towards trustworthiness assurance in the cloud. In: Felici, M. (Ed.), Cyber Security and Privacy. Vol. 182 of Communications in Computer and Information Science. Springer Berlin Heidelberg, pp. 3–15.

Diaz, G., Pardo, J.-J., Cambronero, M.-E., Valero, V., Cuartero, F., 2006. Verification of web services with timed automata. Electronic Notes in Theoretical Computer Science 157 (2), 19–34.

Doelitzscher, F., Ruebsamen, T., Karbe, T., Knahl, M., Reich, C., Clarke, N., 2013. Sun behind clouds - on automatic cloud security audits and a cloud audit policy language. International Journal on Advances in Networks and Services 6 (1–2), 1–16.

Duncan, A., Creese, S., Goldsmith, M., June 2012. Insider attacks in cloud computing. In: Proc. of IEEE TrustCom 2012. Liverpool, UK.

Ernst, M., Cockrell, J., Griswold, W., Notkin, D., May 1999. Dynamically discovering likely program invariants to support program evolution. In: Proc. of ICSE 1999. Los Angeles, California, USA.

Gai, K., Qiu, M., Zhao, H., 2017. Cost-aware multimedia data allocation for heterogeneous memory using genetic algorithm in cloud computing. IEEE Transactions on Cloud Computing PP (99), 1–1.

Gai, K., Qiu, M., Zhao, H., Tao, L., Zong, Z., 2016. Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing. Journal of Network and Computer Applications 59, 46 – 54.
URL http://www.sciencedirect.com/science/article/pii/S108480451500123X

Grobauer, B., Walloschek, T., Stocker, E., March-April 2011. Understanding cloud computing vulnerabilities. IEEE Security & Privacy 9 (2), 50–57.

Herrmann, D., 2002. Using the Common Criteria for IT security evaluation. Auerbach Publications.

Kaluvuri, S., Koshutanski, H., Di Cerbo, F., Mana, A., June–July 2013. Security assurance of services through digital security certificates. In: Proc. of ICWS 2013. Santa Clara, CA, USA.

Kourtesis, D., Ramollari, E., Dranidis, D., Paraskakis, I., 2010. Increased reliability in SOA environments through registry-based conformance testing of web services. Production Planning & Control 21 (2), 130–144.

Kunz, I., Stephanow, P., March 2017. A process model to support continuous certification of cloud services. In: Proc. of AINA 2017. Taipei, Taiwan.

Li, Z., Liao, L., Leung, H., Li, B., Li, C., 2017. Evaluating the credibility of cloud services. Computers & Electrical Engineering 58, 161 – 175.

Lins, S., Grochol, P., Schneider, S., Sunyaev, A., 2016a. Dynamic certification of cloud services: Trust, but verify! IEEE Security & Privacy 14 (2), 66–71.

Lins, S., Schneider, S., Sunyaev, A., 2016b. Trust is good, control is better: Creating secure clouds by continuous auditing. IEEE Transactions on Cloud Computing PP (99), 1–1.

Lins, S., Teigeler, H., Sunyaev, A., June 2016c. Towards a bright future:ÃĆ enhancing diffusion of continuous cloud service auditing by third parties. In: Proc. of ECIS 2016. Istanbul, Turkey.

Luna, J., Taha, A., Trapero, R., Suri, N., 2017. Quantitative reasoning about cloud security using service level agreements. IEEE Transactions on Cloud Computing PP (99).

Maalej, A., Hamza, M., Krichen, M., Jmaiel, M., March 2013. Automated significant load testing for ws-bpel compositions. In: Proc. of IEEE ICSTW 2013. Luxembourg.

Merten, M., Howar, F., Steffen, B., Pellicione, P., Tivoli, M., 2012. Automated inference of models for black box systems based on interface descriptions. In: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. Vol. 7609 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 79–96.

Modic, J., Trapero, R., Taha, A., Luna, J., Stopar, M., Suri, N., 2016. Novel efficient techniques for real-time cloud security assessment. Computers & Security 62, 1 – 18.

Munoz, A., Mãna, A., June 2013. Bridging the gap between software certification and trusted computing for securing cloud computing. In: Proc. of IEEE SERVICES 2013. Santa Clara, CA, USA.

Nunez, D., Fernandez-Gago, C., Luna, J., 2016. Eliciting metrics for accountability of cloud systems. Computers & Security 62, 149 – 164.

OpenStack Foundation, July 2016. OpenStack Security Guide. `http://docs.openstack.org/security-guide/`.

OPTET project, 2013. D3.1 Initial concepts and abstractions to model trustworthiness. `http://www.optet.eu/project/`.

Rak, M., Suri, N., Luna, J., Petcu, D., Casola, V., Villano, U., Dec 2013. Security as a service using an sla-based approach via specs. In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science. Vol. 2. pp. 1–6.

Ravindran, K., January 2013. Model-based engineering methods for certification of cloud-based network systems. In: Proc. of COMSNETS 2013. Bangalore, India.

Spanoudakis, G., Damiani, E., Mana, A., October 2012. Certifying services in cloud: The case for a hybrid, incremental and multi-layer approach. In: Proc. of HASE 2012. Omaha, NE, USA.

Stephanow, P., Khajehmoogahi, K., March 2017. Towards continuous security certification of software-as-a-service applications using web application testing techniques. In: Proc. of AINA 2017. Taipei, Taiwan.

Stephanow, P., Srivastava, G., Schutte, J., June-July 2016. Test-based cloud service certification of opportunistic providers. In: Proc. of IEEE CLOUD 2016. San Francisco, CA, USA.

Sunyaev, A., Schneider, S., February 2013. Cloud services certification. CACM 56 (2), 33–36.

Trapero, R., Modic, J., Stopar, M., Taha, A., Suri, N., 2017. A novel approach to manage cloud security sla incidents. Future Generation Computer Systems 72, 193–205.

Ye, L., Zhang, H., Shi, J., Du, X., December 2012. Verifying cloud service level agreement. In: Proc. of GLOBECOM 2012. Anaheim, CA, USA.

Yeung, W., December 2006. Mapping ws-cdl and bpel into csp for behavioural specification and verification of web services. In: Proc. of ECOWS 2006. Zurich, Switzerland.

```
INPUT
m:=model
T:=execution traces T_i

OUTPUT
[bool_value,r]

GLOBAL VARS
r_partial, r_new, r_broken: set of inconsistencies

MAIN
for each trace T_i ∈ T
    matching_trace(T_i, m);
for each path p_j ∈ m
    if (p_j, any_value) ∉ r_partial∪r_new{
      T_i=generate(p_j);
      matching_trace(T_i, p_j)≠[1,(T_i, p_j)] ? r_broken ∪= {(−, p_j)};
    }
r=r_partial∪r_new∪r_broken;
return r==∅ ? [1,∅] : [0,r];

MATCHING_TRACE(T_i, m)
for each path p_j ∈ m {
    if (p_j ≡ T_i)
      return [1,{(T_i, p_j)}];
    if (p_j ⊂ T_i)
      r_new ∪= {(T_i, −)};
    if (T_i ⊂ p_j )
      r_partial ∪= {(T_i, p_j)};
}
return ∅;
```

Figure A.16: Algorithm *MSV* for model structure verification.

Appendix A. Model Structure Verification (*MSV*)

Figure A.16 illustrates the pseudocode of our algorithm for model structure verification (*MSV*). *MSV* receives as input a system model m and a set $\mathcal{T}$ of traces T_i (both real and synthetic). It is based on a consistency function $\equiv$ defined as follows.

Definition Appendix A.1 (Consistency Function $\equiv$). *Given a trace T_i=$\langle a_1,\ldots,a_n\rangle$∈$\mathcal{T}$ and p_j=$\langle l_0,\ldots,l_n\rangle$ in m∈M, T_i≡p_j iff $\forall a_k$∈T_i, $\exists$ a transition (l_{k-1},l_k)∈E annotated with action a_p s.t. a_k and a_p refer to the same service/operation.*

MSV sequentially matches all traces with paths (first **for each** cycle in **Main**), using function **Matching_Trace**, and returns as output matching results [*bool_value,r*]. Function **Matching_Trace** receives as input the system model m and a trace T_i, and checks whether trace T_i can be matched to a path p_j∈m($p_j \equiv T_i$). The algorithm sequentially analyses all paths (**for each** cycle in **Matching_Trace**) until a match is found (**return** $[1,(T_i, p_j)]$). Otherwise, a matching failure of class partial or new path discovery is generated, that is, there is an inconsistency between the system model and the execution trace that

```
INPUT
m_t:=timed system model
T:=timed traces TT_i

OUTPUT
unmatched_transitions:={(l_i,l_j)}

MAIN
unmatched_transitions:=∅;
for each TT_i ∈ T{
   if matching_trace(TT_i, m_t)==[1,{(TT_i, p_j)}]
     if (TT_i ≢_t p_j)
       for each transition (l_i,l_k) ∈ p_j
           unmatched_transitions ∪= {(l_i,l_k): λ_t((l_i,l_k)) is not satisfied}
}
return unmatched_transitions;
```

Figure B.17: Algorithm TCV for time constraint verification.

could affect an existing certification process and invalidate an issued certificate. Each inconsistency is added to the relevant set (either $r_{new} \cup= \{(T_i, -)\}$ or $r_{partial} \cup= \{T_i, p_j\}$). When all traces have been checked (first **for each** cycle in **Main**), paths p_j that have not been already matched or for which an error was not found are checked for broken existing paths (second **for each** in **Main**). A trace for each of these paths p_j is created (**generate**(p_j)) and matched (**Matching_Trace**(T_i, p_j)). If **Matching_Trace** does not return (T_i, p_j) a broken existing path inconsistency is raised and ($-$, p_j) added to set r_{broken}.

Appendix B. Time Constraint Verification (TCV)

Figure B.17 illustrates the pseudocode of our algorithm for time constraint verification (TCV). TCV receives as input m_t and a set $\mathcal{T}$ of timed traces TT_i. It is based on a time consistency function $\equiv_t$ defined as follows.

Definition Appendix B.1 (Time Consistency Function $\equiv_t$). *Given a trace $TT:=\langle\{o_1,t_1\},\{o_2,t_2\},\ldots,\{o_n,t_n\}\rangle \in \mathcal{T}$ (Definition 3.5) and a path $p=\langle l_0,\ldots,l_n\rangle$ in $m_t \in M$ (Definition 3.4) with annotations $\lambda_t((l_i,l_j))$, such that $TT \equiv p$, $TT \equiv_t p$ iff $\forall$ time annotation $\lambda_t((l_i,l_j))$ associated with p, the corresponding TT is such that the difference $(t_j - t_i)$ between timestamp t_j corresponding to l_j and t_i corresponding to l_i satisfies $\lambda_t((l_i,l_j))$.*

For each timed trace (first **for each** cycle in **Main**), TCV first checks whether it matches a path in m_t (function **Matching_Trace** of algorithm MSV in Figure A.16). If yes, $\{(TT_i,p_j)\}$ is retrieved. It then matches TT_i with p_j according to $\equiv_t$. If for all pairs $TT_i \equiv_t p_j$, then an empty list (success) is returned; otherwise a list of transitions $(l_i,l_k) \in p_j$ that do not satisfy the corresponding annotation (failure) is returned.

```
INPUT
m_prob:=probabilistic timed system model
T:=timed traces TT_i

OUTPUT
unmatched_transitions:={(l_i,l_j)}

MAIN
unmatched_transitions:=∅;
for each TT_i ∈ T{
    if matching_trace(TT_i, m_prob)==[1,{(TT_i, p_j)}]
      for each (l,l_k)∈p_j
          count_traversals[(l,l_k)]+=1;
}
for each p_j ∈ m_prob
    for each transition (l,l_k) ∈ p_j
        prob=count_traversals[(l,l_k)]/∑_s count_traversals[(l,l_s)];
    if ({TT_i} ≢_prob p_j)
      unmatched_transitions ∪= {(l,l_k): λ_prob((l,l_k)) is not satisfied by prob}
return unmatched_transitions;
```

Figure C.18: Algorithm *PCV* for probability constraint verification

Appendix C. Probability Constraint Verification (*PCV*)

Figure C.18 illustrates the pseudocode of the algorithm for probability constraint verification (*PCV*). *PCV* receives as input m_{prob} and a list $\mathcal{T}$ of timed traces TT_i belonging to the time window under evaluation. It is based on a probability consistency function $\equiv_{prob}$ defined as follows.

Definition Appendix C.1 (Probability Consistency Function $\equiv_{prob}$). *Given a set of timed traces TT_j (Definition 3.5) observed in a time window Δt, insisting on path p in $m_{prob} \in M$ (Definition 3.6) with annotations $\lambda_{prob}((l_i,l_{i+1}))$ and such that $TT_j \equiv p$, $\{TT_j\} \equiv_{prob} p$ iff $\forall$ probability annotation $\lambda_{prob}((l_i,l_{i+1}))$ associated with p, the corresponding timed traces TT are such that the frequency of executing p satisfies $\lambda_{prob}((l_i,l_{i+1}))$.*

For each timed trace (first **for each** cycle in **Main**), *PCV* first checks whether the trace matches a path in m_{prob} (function **Matching_Trace** of algorithm *MSV* in Figure A.16). If yes, a counter of transition traversals is increased (**count_traversals**) for each transition in the path (second **for each** cycle in **Main**). It then evaluates the probability constraints assigned to each path in m_{prob} ($\{TT_i\} \not\equiv_{prob} p_j$). To this aim, it calculates the probability of traversing a transition for each state $l \in L$ ($\sum_s$ **count_traversals**$[(l,l_s)]$). If the constraint is satisfied, it is considered to be a positive match, otherwise, not matching transitions $(l,l_k) \in p_j$ are added to a list of negative matches. Algorithm *PCV* then produces as output the list of negatively matched transitions (empty list means no negative traces were found).

```
INPUT
m_r:=trustworthy model

OUTPUT
unmatched_transitions:={(l_i,l_j)}

MAIN
unmatched_transitions:=∅;
for each transition (l_i,l_j)∈m_r{
    if (check_configuration(λ_c((l_i,l_j))) == false)
      unmatched_transitions ∪= {(l_i,l_j)};
}
return unmatched_transitions;
```

Figure D.19: Algorithm CCV for configuration constraint verification

Appendix D. Configuration Constraint Verification (CCV)

Figure D.19 illustrates the pseudocode of the algorithm for configuration constraint verification (CCV). CCV takes as input the trustworthy system model m_r and verifies the configuration properties defined over the model transitions. For each transition, it calls function **Check_Configuration** and depending on the property to be verified it invokes the appropriate mechanism (e.g., check configuration files, monitor configuration-dependent execution traces). If the configuration constraint does not match the production system configurations, the corresponding transition is added to the list **unmatched_transitions**.

Appendix E. Attack Path Verification (APV)

Figure E.20 illustrates the pseudocode of our algorithm for attack model verification (APV). APV receives as input the system model m, the attack model m_{AT}, and a set $\mathcal{T}$ of attack traces AT_i. APV sequentially matches all traces with paths in m (**for each** cycle in **Main**), using function **Matching_Trace**, and returns as output matching results [$bool\_value$,r_{attack}]. Function **Matching_Trace** receives as input the system model m and an attack trace AT_i, and checks whether trace AT_i can be matched to a path $p_j \in m (T_i \equiv p_j)$. The algorithm sequentially analyses all paths (**for each** cycle in **Matching_Trace**); when a match is found, meaning that an attack has been successfully executed, the pair (AT_i,p_j) is added to the attack set r_{attack}.

INPUT
m:=system model
m_{AT}:=attack model
$\mathcal{T}$:=attack traces AT_i

OUTPUT
r_{attack}: set of successful attacks

MAIN
for each *trace* $AT_i \in \mathcal{T}$
  matching_trace(AT_i, m);
return r_{attack}==$\emptyset$? [1,$\emptyset$] : [0,r_{attack}];

MATCHING_TRACE(AT_i, m)
for each *path* $p_j \in m$
  if ($AT_i \equiv p_j$) {
    $r_{attack} \cup= \{(AT_i, p_j)\}$;
    exit(0);
  }

Figure E.20: Algorithm APV for attack model verification.