A Continuous Certification Methodology for DevOps

Marco Anisetti
Università degli Studi di Milano
Milano, Italy
marco.anisetti@unimi.it

Claudio A. Ardagna
Università degli Studi di Milano
Milano, Italy
claudio.ardagna@unimi.it

Filippo Gaudenzi
Università degli Studi di Milano
Milano, Italy
filippo.gaudenzi@unimi.it

Ernesto Damiani
Khalifa University
Abu Dhabi, UAE
ernesto.damiani@kustar.ac.ae

ABSTRACT

The cloud paradigm has revolutionized the way in which software systems are designed, managed, and maintained. With the advent of the microservice architecture, this trend was brought to the extreme, pushing the whole software development process towards unification of software development (Dev) and software operation (Ops). This rapid evolution has not immediately found counterparts in assurance techniques, where the evaluation of the non-functional behavior of a software system and of the software development process are completely decoupled. In this paper, we put forward the idea that next-generation assurance techniques, and more specifically certification techniques, must evaluate a software system throughout the whole development process. To this aim, we define a continuous certification scheme for DevOps that evaluates the software artifacts produced at each stage of the development process. We then present the assurance framework managing our certification scheme and experimentally evaluate the continuous certification scheme in a real DevOps scenario.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; *Extra-functional properties*; Development frameworks and environments; • **Security and privacy** → *Software security engineering*.

KEYWORDS

Assurance, Certification, DevOps

ACM Reference Format:
Marco Anisetti, Claudio A. Ardagna, Filippo Gaudenzi, and Ernesto Damiani. 2019. A Continuous Certification Methodology for DevOps. In *11th International Conference on Management of Digital EcoSystems (MEDES '19), November 12–14, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297662.3365827

1 INTRODUCTION

A cloud environment is a complex ecosystem where computational resources and services can be provisioned on demand, adapted based on events, and refactored as needed. The undebatable advantages of cloud environments come at the price of an increased uncertainty and reduced transparency of their processes, where application developers have little insights about their overall operation and behavior. With the advent of the microservice architecture, the uncertainty and lack of transparency become even worse because cloud applications orchestrate a panoply of (micro)services under the control of different providers.

Enabling assurance techniques [7], including certification, has become a key requirement for the widespread diffusion of modern software systems, where verification of non-functional properties like security, privacy and dependability is a must-have feature for end users. According to Denney and Fisher [13], software certification is the process that "*demonstrates the reliability and safety of software systems in such a way that it can be checked by an independent authority with minimal trust in the techniques and tools used in the certification process itself.*" Only recently, this definition has been revisited to accomplish new requirements introduced by the evolution of the software development towards IT services, cloud environments, and microservices. Certification has been moving from manual, after-deployment processes, to semi-automatic tools for run time evaluation. New certification schemes for the cloud have been defined [1, 3, 14, 15, 19], where the involvement of the certification authority is weakened, and the importance of semi-automatic and trustworthy techniques and tools raises. Notwithstanding their huge potential, such approaches still prove a poor fit for modern systems that mix cloud applications and microservices. We argue that in the era of the unification of software development (Dev) and software operation (Ops), we need to depart from software verification approaches where the evaluation of the non-functional behavior of the software system is completely decoupled from the evaluation of the software development process. Cloud platforms evolution toward service miniaturization is also paving the way to lightweight and configurable certification that needs to be aligned with the Continuous Integration (CI) and Continuous Delivery (CD) development model. We then refine the notion of *continuous certification* as a multi-step software system verification process that is "synchronous" with and driven by the software development process. It specifies the hooks for certification, specifies certification requirements, supports the deployment of the verification mechanisms, and manages and adapts their execution when moved between the development stages and the operation system. The contribution of the paper is manifold. We first define a continuous certification scheme for DevOps. We then implement the corresponding framework for service-based system certification. We finally test the certification scheme in a real scenario for the verification of security properties. To this aim, we use DevSecOps manifesto (http://www.devsecops.org/), explicitly mentioning the need for "Compliance Operations", as a source of requirements for a DevOps process and corresponding system under development. The paper is organized as follows. Section 2 discusses the related work. Section 3 discusses the notion of continuous certification applied to a software development process and the requirements it poses on certification schemes. After introducing our reference scenario (Section 4), Sections 5 and 6 discuss our certification scheme for DevOps and its instantiation. Section 7 presents our assurance framework. Section 8 experimentally evaluates the certification-by-design concept in a real environment. Section 9 gives our conclusions.

2 RELATED WORK

The advent of DevOps as a set of software development practices that increase the ability of organizations to deliver applications and services at high velocity, coupled with the automation introduced by Continuous Integration (CI) and Continuous Delivery (CD) paradigm, require a re-thinking of assurance techniques that only verify a system once deployed in staging or production. This is particularly critical for security assurance, where security requirements must be specified since the system design [10] and drive security testing as part of CI [20]. DevSecOps is a step in this direction and incorporates security practices in the classic DevOps process. Mohan et al. [18] report a survey to identify the main aspects of this trend. Shain et al. [21] also report a systematically literature review on continuous practices from 2006 to 2014. Mattetti et al. [16] present a framework that can be applied to evaluate the security of Linux containers and their workloads by defining and describing rules and expected activities. Mohan et al. [17] describe the experience made in IBM on deployment automation of 5 Business Intelligence projects with specific security concerns: logging, separation of rules, enforcement of accesses, audit. Debroy et al. [12] describe the automation of a web application testing on a CI/CD, starting from a series of non-functional requirements. Bass et al. [9] go one step forward investigating how to build a trustworthy pipeline where integrity and security are the first concerns. Ullah et al. [24] face the problem of providing a secure pipeline by analyzing security requirements on code, data centers, and CI server, which are the main components of a continuous deployment pipeline.

Recent service certification schemes [1–3, 5, 7, 22, 23] implemented a continuous process aimed to continuously collect evidence that infrastructures and/or applications consistently demonstrate one or more non-functional properties. The role of certification tools (*controls* in the following) becomes fundamental, being the source of information for certificate issuing. The accredited lab, the entity responsible for manual verification, is substituted by semi-automatic frameworks that recurrently execute controls to collect/aggregate evidence, which is reported back to the central certification authority for the final evaluation. We note that, given the high dynamics of service life cycle, recent certification techniques departed from the assumption of a full involvement of the certification authority in the management of certification processes. Rather, the certification authority provides (signed) certification model templates specifying the certification process as a computation to be performed on observable evidence, while semi-automatic techniques are used to execute each computation [4].

Some approaches focused on the certification of the development process only [8, 25], just few took the specific system under development into account. Darwish at al. [11] for instance developed a fuzzy-based fusion approach to combine process and software metrics. To the best of our knowledge, no certification schemes collect and aggregate evidence on system behavior as a part of the development process used for its implementation.

In this context, the widely used notion of "by-design", which in most of the cases is associated with formal verification practices,[1], can be also seen as coupled with the software development process, reinforcing the tight relation between the development process and the implemented system. For instance, the EU General Data Protection Regulation (GDPR) mandates *privacy-by-design* and *security-by-design* in the development of systems, meaning that privacy and security properties should be verified since requirements' definition and across all development stages.

In this paper, we put forward the idea that certification processes must move from traditional verification of service-based systems to a notion of continuous certification, where certification requirements are imposed during the development process and non-functional property verification is planned since design time.

3 CONTINUOUS CERTIFICATION

A modern certification scheme must evaluate a target system along all stages of the development process, supporting the concept of *continuous certification*. Continuous certification requires a multi-step software system verification process that is driven by the development process, the certification requirements, and the properties to be certified on the target system. Certification activities need to target all (intermediate) software artifacts and bind collected evidence to all development stages, linking evidence back to certification requirements. This goal can be only achieved by "shifting to left" the non-functional (e.g., security) activities that usually has been thought as a "bolt-on" process after the provisioning of the application. More in detail, the notion of continuous certification introduces new requirements on certification schemes that are summarized as follows.

- **Development process modeling:** the development process, regardless of how it is implemented, must be part of the certification model that is under the responsibility of the certification authority.
- **Stage-by-stage evaluation:** The certification model must describe certification requirements and the corresponding controls used to collect relevant evidence at each development stage.

[1]For instance, in the case of "Correctness-by-Design" where one starts from a formal specification and the result is a program satisfying the specification.

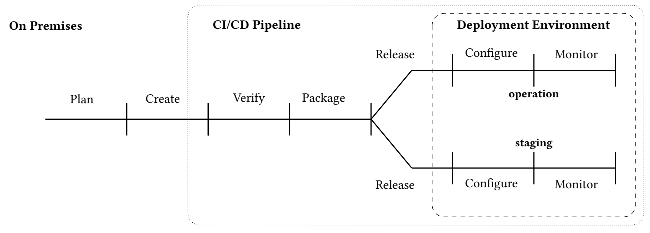

Figure 1: DevOps pipeline for our reference scenario.

- **Trustworthy inspection:** the development process must support trustworthy execution of controls.
- **Continuous verification:** hooks in the development process must be provided to support continuous evidence collection by controls.

Although continuous certification (and corresponding requirements) is applicable to any development process, it provides the best advantages in agile processes, where Continuous Integration (CI) and Continuous Delivery (CD) make the development process and the software it releases a unique conceptual entity. In the reminder of this paper, we then focus on certification of services developed following a DevOps process, defining a new certification scheme for DevOps (Sections 5 and 6). DevOps in fact provides full control and automation of the development steps, and is a natural candidate for supporting continuous certification.

4 REFERENCE SCENARIO

Our reference scenario is a cloud-native application designed as an orchestration of microservices developed following the DevOps principles. The application, developed in the EU project H2020 Toreador (http://toreador-project.eu), implements a methodology providing Big Data Analytics-as-a-Service. It relies on three main services: *i) analytics service* handling the analytics tasks (e.g., cleansing, ingestion, mining), *ii) workflow service* supporting the generation of the analytics workflows as a composition of analytics tasks, and *iii) execution service* triggering workflow execution and managing the interactions with the Big Data ecosystem. The application is designed and implemented using an *API gateway*, which provides advanced functionalities for logging, authentication, and security, on one side, and an interface to interact with front-end components for specific vertical applications, on the other side.

The considered application, being devoted to data management and analytics, must certify security and privacy properties according to requirements collected in relevant regulations (e.g., GDPR). For this reason, we apply our continuous certification approach and define certification requirements at each stage of the development process in Figure 1. The development process is composed of stages that are executed on premise, during the CI/CD pipeline, and in the deployment environment, and drives all certification activities possibly resulting in the issuing of a certificate. An overview of the certification activities done at each DevOps stage is sketched in the following.

- Stage *Plan* designs the application. It also specifies security requirements and metrics to measure performance and quality of service of the application itself.
- Stage *Create* generates the application code, statically verifies it for security reasons, and builds it after the source code is committed.
- Stage *Verify* provides features for functional and non-functional testing of the software.
- Stage *Package* prepares the software artifacts for application deployment, and the corresponding security verification on packaging dependencies.
- Stage *Release* marks the difference between staging and operation pipelines targeting private or public deployments, respectively.
- Stage *Configure* configures the deployment environment, and includes infrastructure-level and container-level security verification.
- Stage *Monitoring* continuously monitors the application to calculate those metrics needed for continuous security verification. We note that stages Release, Configure, and Monitor consider both staging and operation environments.

5 CERTIFICATION PROCESS MODELING

We define a certification process that goes beyond traditional system verification [7], by enriching system certification with the evaluation of the compliance of the development process against non-functional policies and requirements. The certification process is driven by a certification model that triggers a set of assurance activities at each stage of DevOps development process.

Certification models as a way to specify all activities to be carried out in a certification process, from the initial description of assurance activities to the incarnation of these activities into a concrete certification process [4], is a common approach to certification management that has been adopted in many solutions [7]. None of them however incarnates the need of verifying the adherence of the whole development process to policies and requirements; they rather specify a set of (testing or monitoring) activities that are executed on the final system before and after deployment in the in-production environment.

Our certification model $\mathcal{M}$ is a technology- and system-independent abstract representation of the certification process, driving its execution according to the selected policies/requirements. It models the development phases and specifies certification process inputs, including high-level policies/requirements, configurations, and activities for the certification of a property for a given class of target systems. We formalize the certification model $\mathcal{M}$ as follows.

Definition 5.1 ($\mathcal{M}$). *A certification model $\mathcal{M}$ is defined as a 4-tuples of the form $<P_i, T^s_i, C_i, G_i>$, where P_i refers to the i-th DevOps phase triggering the execution of certification activities in C_i, $T^s_i \rightarrow \{T^s_{i,1},\ldots,T^s_{i,n}\} \subseteq T$ is a system artifact, that is, a portion (i.e., $T^s_{i,j}$) of the system (i.e., T) under certification, $C_i \rightarrow \{c_{i,1},\ldots,c_{i,n}\}$ is a list of control types suitable for system certification, and G_i defines the guards that need to be satisfied for triggering a phase transition.*

We note that each C_i specifies a class of controls that points to a set of real controls $\{c_{i,1},\ldots,c_{i,n}\}$ that can be selected for system verification. For instance, class *application web vulnerability* refers to all those controls that provide functionalities for web vulnerability assessment. We also note that guards G_i are defined as a Boolean expressions of conditions of the form $op(\{c,T^s\},EO)$, where op is

P_1=“Plan”
T_1^s={$T_{1,1}^s$=“Software design”}
C_1={c_1=“Check requirements document”}
G_1= =({$c_{1,1}$,T_1^s}, success)

P_2=“Create”
T_2= {$T_{2,1}^s$=“Requirements document”, $T_{2,2}^s$=“Code repository”}
C_2= {$c_{2,1}$=“Check requirements links”, $c_{2,2}$=“Static code analysis”}
G_2= ($c_{2,1}$,$T_{2,1}^s$}, success) $\wedge$ ({$c_{2,2}$, $T_{2,2}^s$, success)

P_3=“Verify”
T_3^s={ $T_{3,1}^s$=“Latest build release”}
C_3={$c_{3,1}$=“Security tests”}
G_3= ({$c_{3,1}$, T_3^s}, success)

P_4=“Package”
T_4^s={$T_{4,1}^s$=“Dependencies”}
C_4={$c_{4,1}$=“Dependencies vulnerability checks”}
G_4= ({$c_{4,1}$ T_4^s}, success)

P_5=“Release”
T_5={$T_{5,1}^s$=“Deployment environment”}
C_5={$c_{5,1}$=“Release procedure checks”}
G_5=({$c_{5,1}$, T_5^s}, success)

P_6=“Configure”
T_6^s={$T_{6,1}^s$=“Deployment Infrastructure”}
C_6={$c_{6,1}$=“Security benchmarking”, c_2=“Pen test”}
G_6=({$c_{6,1}$, T_6^s}, success) $\vee$ ({$c_{6,2}$, T_6^s}, success)

P_7=“Monitor”
T_7^s={$T_{7,1}^s$=“Application”}
C_7={$c_{7,1}$=“Vulnerability scan”, $c_{7,2}$=“Pen test”, $c_{7,3}$=“Execution trace checks”}
G_7=({$c_{7,1}$, T_7^s}, success) $\wedge$ ({$c_{7,2}$, T_7^s}, success) $\wedge$ ({$c_{7,3}$, T_7^s}, success)

Figure 2: An excerpt of the certification model $\mathcal{M}$ for the DevOps process in our reference scenario in Section 4

an operator in {=,≤,≥,<,>}, c is a control, T^s is the target of the control, and EO is the expected output of the execution of control c on target T^s. Guards define conditions that need to be satisfied by the specific verification step in order to proceed with subsequent activities. We note that if a guard is not satisfied, the certification process ended and correction activities are requested to fix the issues discovered by the guard. Our solution is complementary to existing certification solutions and reuses them as much as possible to provide a semi-automatic certification solution.[2]

Figure 2 shows an extract of a certification model for our reference scenario in Section 4.
P_1 refers to stage Plan. C_1 contains a single manual control $c_{1,1}$ ensuring that the requirements document exists and has the format requested to drive the remaining of the process (G_1).
P_2 refers to stage Create. C_2 contains two types of controls: *i)* $c_{2,1}$ verifies whether the requirements document in stage Plan ($T_{2,1}^s$) has been manually annotated to link requirements to the source code components they regulate; *ii)* $c_{2,2}$ statically verifies the source code in the code repository ($T_{2,2}^s$) to identify security issues. In case both $c_{2,1}$ and $c_{2,2}$ succeed (G_2), the following stage is executed.
P_3 refers to stage Verify. C_3 contains a single control $c_{3,1}$ executing ad hoc security testing on the latest version of the build release ($T_{3,1}^s$).
P_4 refers to stage Package. C_4 contains a single control $c_{4,1}$ executing dependency vulnerability checks on the additional packages ($T_{4,1}^s$), if any, required for application packaging.
P_5 refers to stage Release. C_5 contains a single control $c_{5,1}$ verifying the correctness of the release procedure.

[2]The way in which the targets of a verification (i.e., T and T^s) are specified and modeled, and the corresponding verification activities instantiated and executed are out of the scope of this paper. A possible solution has been defined in [3] and used in our experiments.

P_6 refers to stage Configure. C_6 contains two types of controls both having as target to deployment infrastructure ($T_{6,1}^s$): *i)* $c_{6,1}$ performs security benchmarking according to International regulations (e.g., Center for Internet Security – CIS – benchmarking); *ii)* $c_{6,2}$ performs a penetration testing.
P_7 refers to stage Monitor. C_7 contains three types of controls both having as target to system application ($T_{7,1}^s$): *i)* $c_{7,1}$ performs a vulnerability scan; *ii)* c_2 performs a penetration testing; and *iii)* c_3 monitors the executions of the target system when deployed in the operation environment [2].

We note that, according to Figure 1, P_5, P_6, P_7 verify the target system at DevOps stages Release, Configure, and Monitoring, respectively, both in staging and operation environments. This reinforces the concept of a certification process that is strongly coupled with the development pipeline and not only verifies the final deployed system, but rather continuously verifies all artifacts produced since system design.

6 CERTIFICATION PROCESS INSTANTIATION

The certification model in Definition 5.1 is a composite model where certification activities are independently specified for each DevOps stage. These certification activities are then mapped to the DevOps process by multiple instantiation functions λ_i. At a logical level, when the i-th DevOps stage is triggered, the corresponding λ_i is executed and the certification activities in $\mathcal{M}$ instantiated on the relevant system artifact T_i^s (e.g., system code, system build, system package).

We define a vector $\overline{V}$ of instantiation functions λ_i, one for each DevOps stage in $\mathcal{M}$, as follows.

Definition 6.1 (Instantiation Function λ_i). *An instantiation function λ_{P_i} is a function that takes as input a tuple <P_i, T_i^s, C_i, G_i> $\in\mathcal{M}$ and returns as output an instance <$\overline{P}_i$,$\overline{T}_i^s$,$\overline{C}_i$,$\overline{G}_i$> where: i) $\overline{P}_i$ deals with all activities requested for the verification of the i-th DevOps stage, ii) $\overline{T}_i^s$ contains the hooks to the real target (or a part thereof) under certification, iii) $\overline{C}_i$ includes references to the controls that need to be executed on $\overline{T}_i^s$, iv) $\overline{G}_i$ instantiates the guards in $\mathcal{M}$ with the parameters retrieved by executing $\overline{C}_i$ on $\overline{T}_i^s$.*

The result of the execution of the instantiation functions in $\overline{V}$ is an instance $\mathcal{M}^I$ of the certification model $\mathcal{M}$, which drives real certification activities along the DevOps pipeline.[3] We recall that phase Plan in $\mathcal{M}$ requires manual activities; therefore, the instantiation function executes manual checks. We note that the instantiation of the certification activities or, in other words, the execution of instantiation functions λ can be pre-computed, when possible, for performance reasons.

Example 6.1 (Function λ_{mon}). *Let us consider the DevOps Monitor stage only and the relative 4-tuple <P_{mon}, T_{mon}^s, C_{mon}, G_{mon}> $\in\mathcal{M}$. Annotation function $\lambda_{P_{mon}}\in\overline{V}$ is triggered when stage Monitor is reached after a successful verification of guard $\overline{G}_{conf}$. It generates <$\overline{P}_{mon}$, $\overline{T}_{mon}^s$, $\overline{C}_{mon}$, $\overline{G}_{mon}$>, where $\overline{P}_{mon}$ specifies the ordered list of controls to be executed and the environmental information used*

[3]Details on how this instantiation function is implemented is out of the scope of this paper and will be considered in our future work.

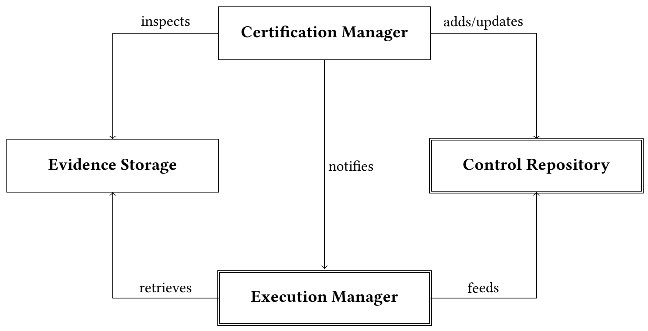

Figure 3: Certification framework. Single line rectangles refer to components deployed on the public cloud; double line rectangles refer to components that can be either deployed on the public cloud or on premises.

to set up the evaluations (see Section 7), $\overline{T}^s_{mon}$ specifies the hooks to the target system (or specific artifacts thereof) deployed on the CI/CD pipeline, $\overline{C}_{mon}$ configures the set of controls $\overline{c}_{mon}$ with all relevant parameters, $\overline{G}_{mon}$ specifies the guards as a Boolean expression of the outputs of each control $\overline{c}_{mon}$.

7 THE CERTIFICATION FRAMEWORK

Our certification scheme is implemented in a certification framework that addresses the following requirements:

- **Limited interference:** the certification framework must minimize the interference with the development process.
- **Minimal impact:** the certification framework must reduce the impact on the application deployment.
- **Full automation:** the certification framework must trigger controls in a fully automatic way.

Our certification framework [6] provides a distributed architecture that integrates evidence collection within a DevOps process. It offers a set of general-purpose controls, as well as a methodology to develop custom controls, to perform the requested security checks. All controls are implemented using containers, scripts skeletons, and wrapping drivers, keeping dependencies confined and preserving isolation between controls.

7.1 Certification Framework Architecture

Figure 3 shows a simplified view our framework architecture implemented as a service in the cloud. It is based on *i) compute nodes* executing evaluation activities on different hooks (*Execution Manager*) to collect evidence used to verify the target system (*Certification Manager*), *ii) storage nodes* storing the certification controls (*Control Repository*) and the results of the evaluation activities (*Evidence Storage*).

More in detail, Certification Manager is the process owner, and implements functionalities to correlate and analyze the collected evidence. It executes a certification model $\mathcal{M}^I$ for a specific DevOps process, collects evidence for system certification, and keeps the control repository up to date. It inspects the evidence collected as the result of the control execution and stored in Evidence Storage to evaluate the relevant guard $\overline{G}_i$. To this aim, it instruments Execution

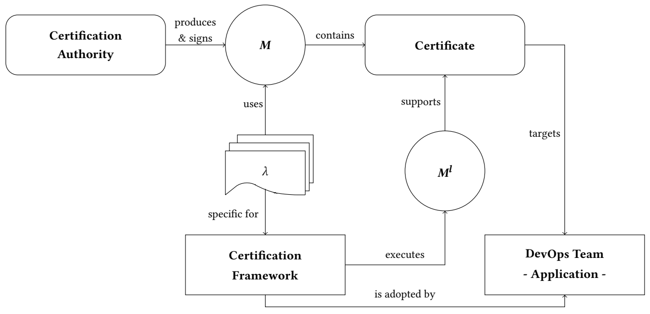

Figure 4: DevOps pipeline for our reference scenario in Section 4.

Manager, which is in charge of executing the certification controls $\overline{C}_i$ on the corresponding targets $\overline{T}^s_i$. General-purpose controls (e.g., channel confidentiality and integrity verification), as well as ad hoc controls usually developed in synergy with the system development, are stored in public or private Control Repositories as versioned scripts. The proposed architecture can scale and migrate with the application development pipeline. It exposes APIs that are called during the different DevOps stages to carry out the certification process.

7.2 Certification Methodology and Execution

Figure 4 presents an overview of the continuous certification methodology implemented in our framework. The certification authority first produces and signs the certification model $\mathcal{M}$, which is later instantiated into a certification instance $\mathcal{M}^I$ by means of instantiation functions λ. The goal of the methodology is to obtain a certificate for the target system, which presents a link to the original model $\mathcal{M}$ and an indirect reference to the certification authority signing it. The instantiation functions λ take a certification model $\mathcal{M}$ as input and generates a certification model instance $\mathcal{M}^I$ as a set of YAML files as output. YAML files are bound to the different stages of the DevOps pipeline and executed by the DevOps team through the framework. Verification activities at each stage are triggered by sending the YAML file to Certification Manager, where it is interpreted and executed. This approach permits to separate the DevOps pipeline from the specific certification activities.

Let us consider certification model $\mathcal{M}$ in Figure 2. Figure 5 presents an excerpt of the YAML file incarnation of tuple $<\overline{P}_{cr}, \overline{T}^s_{cr}, \overline{C}_{cr}, \overline{G}_{cr}> \in \mathcal{M}^I$ for stage Create. Once the YAML file is generated, it is bound to the corresponding stage of the pipeline by the DevOps team. This constitutes the only point of contact between the DevOps pipeline and the certification activities executed by Certification Manager. When stage Create is triggered, the corresponding YAML file incarnation is sent to Certification Manager. Certification Manager first inspects $\overline{P}_i$ of the YAML in order to retrieve environmental information, such as available Execution Managers and Control Repositories for the specific stage (e.g., "ExeMan1" and "PublicRepo" in Figure 5). Then, the portion of the YAML file specifying target $\overline{T}^s_i$ and controls $\overline{C}_i$ is used to instrument Execution Manager with *i)* the controls to be executed, each one with

```
# (P_create) env information
Create:
  −env:
    repo:
      − PublicRepo:
          url:...
          credentials:***
    ExeMan:
      − ExeMan1:
          url:...
          certificate:***

# (T_create) target definitions
  −targets:
    − softwarerepo:
        rootpath:....
        executor: ExeMan1
    − requdoc:
        rootpath:....
        executor: ExeMan1
```

```
# (C_create) controls definition
  − controls:
    − id: RequirementsChecks
          repo: PublicRepo
          parameters:
            reqdocpath=...
    − id: StaticCodeAnalysis
          repo: PublicRepo
          parameters:
            language=nodejs ...

# (R_create) rule definition
  − rule:
      connectors: and
      controls:
        − RequirementsChecks
            target: softwarerepo
            EO: success
            op: equal
        − StaticCodeAnalysis
            target: requdoc
            EO: success
            op: equal
```

Figure 5: An excerpt of the YAML file in $\mathcal{M}^I$ for stage Create of $\mathcal{M}$ in Figure 2.

the relevant parameters (e.g., "language: nodejs" for control StaticCodeAnalysis in Figure 5), *ii)* the repository to be contacted to obtain the container wrapping each control. Execution Manager retrieve the controls from Control Repository and executes them. The results of their execution are stored in Evidence Storage and used by Certification Manager to verify $\overline{G}_i$. If the evaluation of the guard is successful the DevOps pipeline proceeds with the following stage; otherwise, it stops and a certification error is returned.

We remind that a DevOps process is a continuous process, where each stage can be triggered for a number of different reasons, further triggering the subsequent stages until the deployment in operation/staging. For instance, a new functionality may require a refinement of the CI/CD pipeline since stage Plan, while a new security recommendation on the infrastructure/container may trigger the process from stage Configuration. In addition, since the certification process is continuous itself, an update on some of the controls in Control Repository will automatically trigger their re-execution on the pipeline even if no DevOps activities are requested. For instance, in case a new security threat for NodeJS in our reference scenario in Figure 5 is disclosed, control StaticCodeAnalysis is updated in the repository and Certification Manager contacts ExecutionManager1 to re-execute the updated control.

Finally, hook and control configurations may change between different DevOps stages. For instance, a complete vulnerability assessment can be executed during stage Monitoring of the staging environment, while it could just control the most relevant vulnerabilities in stage Monitoring of the operation environment (e.g., for performance reasons). Additional hooks and controls can also be defined to tackle specific deployment platforms like Amazon AWS or Microsoft Azure, for instance, monitoring the infrastructure during the configuration stage by connecting to Azure Security Center.

8 EXPERIMENTAL EVALUATION

We experimentally evaluate our certification scheme in the reference scenario in Section 4.

8.1 Experimental setup

For simplicity but with no lack of generality, we considered the *API gateway* in our reference scenario (Section 4) as the target of certification and *confidentiality in transit* as the security property to be certified on it. The API gateway is based on *Moleculer Web API gateway* (https://github.com/molecularjs/moleculer-web), the API gateway of *Moleculer*, a microservice framework developed in NodeJS (https://moleculer.services/). We implemented the corresponding DevOps pipeline in Gitlab for CI/CD, and integrated its declarative specification in our certification framework. Openshift 3.7 was selected as the operation environment just for the scope of this experiments, since it provides a free account for testing purposes.

8.2 Certification case study

We refined the certification model $\mathcal{M}$ in Figure 2 to fit the API gateway peculiarities. First, to setup the certification process, we specified a declarative description of the security requirements document (Figure 6(a)), where the link to the target source code is specified. We then implemented the instantiation functions λ_i to generate $\mathcal{M}^I$; λ_i receive the requirements doc and $\mathcal{M}$ as input, and produces the corresponding YAML files.[4] Annotation functions can be executed at stage Plan to pre-compute the YAML files for all stages, or, alternatively, at run time during each single stage. Figure 6(b) shows a portion of the complete YAML file generated by λ_{mon} for stage Monitoring, where control *https-robustness* is defined. We note that the YAML file in Figure 6(b) extends the one in Figure 5 with a number of additional activities and configuration details. For instance, two separate test cases are listed as part of this control ("c#73" and "c#75") and a list of additional parameters is provided. In case of test "73", the target port is 443 and, being a monitoring control, the execution is scheduled every 24 hours.

8.3 Performance Evaluation and Discussion

Different sets of (custom) controls have been defined in $\mathcal{M}^I$ for each stage of DevOps as follows: *i)* requirement check (custom) and static code inspection based on sonarqube [*Create*], *ii)* security test (custom) [*Verify*], *iii)* dependency check using a control based on audit for NodeJS [*Package*], *iv)* configuration check based on kube-hunter [*Configure*], *v)* HTTPS robustness check based on nmap and NodeJS penetration test (custom) [*Monitoring*].

Figure 7 presents the execution time of the above controls for the API gateway verification process carried out in Gitlab. These values have been obtained as an average of 10 executions. The entire pipeline with all controls executed took around 465s. The most expensive stage was stage Create (241s) and, in particular, the control static code analysis that took 89s; the less expensive was stage Release (0s) with no controls to be executed. In addition to

[4]We note that the reason why YAML files have been used to represent certification models is twofold: *i)* YAML is the *de-facto* standard for the representation of DevOps stages, *ii)* YAML provides a human-readable format that supports validation activities that are not completely automatic.

```
{
  "Specifications": [
   {
   "SrcPath": "CWD/../
             examples/ssl",
   "SrcFiles": "index.js",
   "CompulsoryProperties": [
      {
      "Name": "Channel−Safety",
      "Type": "https−robustness",
      "OccurencesMin": "1"
      }
      ]
    }
   ],
   ...
   "Export": {
      "Path": "CWD",
      "File": "AgentConfig.yaml"
   }
}
```

(a)

```
# (C Monitor) controls definition
− controls:
  − id: https−robustness
    repo: PublicRepo
    parameters:
      url=***
      ...
    mapping:
      − 'c#73': '0'
      − 'c#75': '1'
    tests:
     − control: 73
       name: Encrypted−channel
       execution_cluster: 1
       interval_detail:
         id: 0
         periods: scheduled
         every: 86400
       testcase:
         config:
           port: 443
       running: {}
       target: 165
       target_detail: mol_web_stage_os
     ...
```

(b)

Figure 6: An excerpt of (a) requirements definition and (b) a YAML file generated for requirements corresponding to stage Monitor.

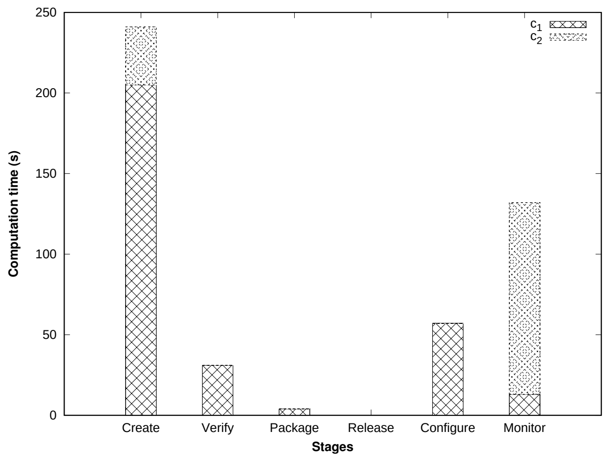

Figure 7: Control execution time for API gateway verification process.

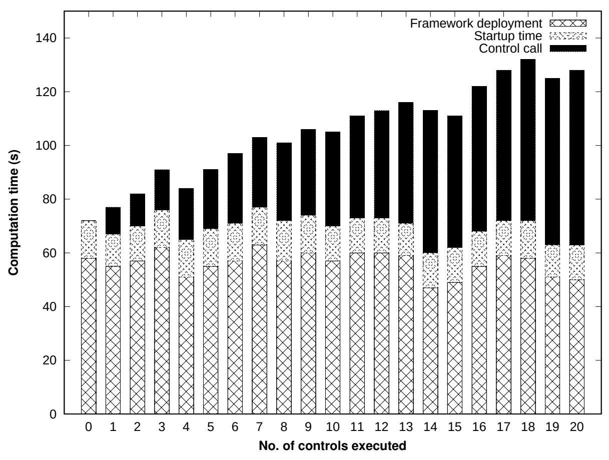

Figure 8: Average overhead (computation time requested for framework deployment, startup, and control call) varying the number of executed controls from 0 to 20.

the control execution time in Figure 7, additional 89*s* were needed for pipeline settings, initialization, and deploy in operation, which brought to a total execution time of 554*s*. We note that, on top of this, the $\mathcal{M}^I$ generation time, which have been pre-calculated offline during stage Plan, took 2.5*s* on a laptop based on Intel Core i5-4310M with 16GB of RAM and 250GB SSD. The execution of the controls revealed 131 vulnerabilities, three with moderate impact and two with high impact on the software code, and one with high impact at Openshift configuration level. The latter vulnerability permitted to access to pod's service account token, giving an attacker the possibility to use the server APIs (privilege escalation).

Figure 8 presents the total overhead introduced by our framework increasing the number of executed controls from 0 to 20. The overhead is 72*s* on average without control execution time, which is less than 13% of the entire pipeline execution time (554*s* on average). In the worst case of 20 controls, the overhead raises to 128*s* on average. Our experimental evaluations shows that *i)* the computational effort requested by each control is dominated by the control complexity, *ii)* the overhead of adopting our framework is negligible compared to the time of executing the controls, *iii)* the overhead for framework deployment and startup is independent by the number of controls, while the control call is linear in the number of controls.

To conclude, our framework provides the following advantages with respect to an approach implementing the same controls directly in the pipeline. First, the controls provided by our framework are continuously updated to accomplish new threats/vulnerabilities. Second, new controls are continuously added to the framework repository to cope with new threats/vulnerabilities, supporting a continuous certification process. Third, embedding scripts in the pipeline can become a time-consuming effort for the DevOps team, which is reduced when our semi-automatic approach is adopted.

9 CONCLUSIONS

We presented a new certification scheme for DevOps that evaluates the software artifacts produced at each stage of the development process. The proposed scheme supports the concept of *continuous certification* for DevOps, where a system is developed with certification in mind. This paper leaves space for future work. First of all, the definition of a complete chain of trust grounded on trustworthy instantiation functions. Then, the refinement of our certification scheme towards any development processes.

ACKNOWLEDGMENTS

This work has received funding from European Union's Horizon 2020 research and innovation programme under grant agreement No 830927 and by the program "piano sostegno alla ricerca 2015-17" funded by Università degli Studi di Milano. We also would like to thank Lorenzo Girardin for his help in the experimental evaluation.

REFERENCES

[1] Marco Anisetti, Claudio Ardagna, and Ernestoi Damiani. 2011. Fine-Grained Modeling of Web Services for Test-Based Security Certification. In *Proc. of SCC 2011*. Washington, DC, USA.

[2] Marco Anisetti, Claudio Ardagna, Ernesto Damiani, Nabil El Ioini, and Filippo Gaudenzi. 2018. Modeling time, probability, and configuration constraints for continuous cloud service certification. *COSE* 72 (2018), 234–254.

[3] Marco Anisetti, Claudio Ardagna, Ernesto Damiani, and Filippo Gaudenzi. 2017. A semi-automatic and trustworthy scheme for continuous cloud service certification. *IEEE TSC* (2017).

[4] Marco Anisetti, Claudio Ardagna, Ernesto Damiani, Filippo Gaudenzi, and Roberto Veca. 2015. Toward Security and Performance Certification of OpenStack. In *Proc. of IEEE CLOUD 2015*. New York, NY, USA.

[5] Marco Anisetti, Claudio Ardagna, Ernesto Damiani, and Gianluca Polegri. 2018. Test-Based Security Certification of Composite Services. *ACM TWEB* 13, 1 (2018), 3.

[6] Marco Anisetti, Claudio Agostino Ardagna, Filippo Gaudenzi, and Ernesto Damiani. 2016. A certification framework for cloud-based services. In *Proc. of ACM SAC*. 440–447.

[7] Claudio Ardagna, Rasool Asal, Ernesto Damiani, and Quang H. Vu. 2015. From Security to Assurance in the Cloud: A Survey. *ACM CSUR* 48, 1 (2015), 2:1–2:50.

[8] Fauziah Baharom, Jamaiah Yahaya, Aziz Deraman, and A Razak Hamdan. 2011. SPQF: software process quality factor. In *Proc. of ICEEI 2011*. Bandung, Indonesia.

[9] Len Bass, Raóph Holz, Paul Rimba, An B. Tran, and Liming Zhu. 2015. Securing a Deployment Pipeline. In *Proc. of RELENG*. Florence. Italy.

[10] Kim Carter. 2017. Francois Raynaud on DevSecOps. *IEEE Software* 34, 5 (2017), 93–96.

[11] Saad M Darwish. 2016. Software test quality rating: A paradigm shift in swarm computing for software certification. *Knowledge-Based Systems* 110 (2016), 167–175.

[12] Vidroha Debroy, Lance Brimble, Matthew Yost, and Archana Erry. 2018. Automating Web Application Testing from the Ground Up: Experiences and Lessons Learned in an Industrial Setting. In *Proc. of IEEE ICST 2018*. Vasteras, Sweden.

[13] Ewen Denney and Bernd Fischer. 2005. Software certification and software certificate management systems. (2005). https://ti.arc.nasa.gov/m/pub-archive/archive/1095.pdf.

[14] Colins.S. Gordon, Michael.D. Ernst, Dan Grossman, and Matthew J. Parkinson. 2017. Verifying Invariants of Lock-Free Data Structures with Rely-Guarantee and Refinement Types. *ACM TOPLAS* 39, 3, Article 11 (2017), 11:1–11:54 pages.

[15] Sebastian Lins, Pascal Grochol, Stephan Schneider, and Ali Sunyaev. 2016. Dynamic Certification of Cloud Services: Trust, but Verify! *IEEE S&P* 14, 2 (2016), 66–71.

[16] Massimiliano Mattetti, Alexandra Shulman-Peleg, Yair Allouche, Antonio Corradi, Shlomi Dolev, and Luca Foschini. 2015. Securing the infrastructure and the workloads of linux containers. In *Proc. of IEEE CNS 2015*, Vol. San Francisco, CA, USA.

[17] Vaishnavi Mohan, Lotfi ben Othmane, and Andre Kres. 2018. BP: Security Concerns and Best Practices for Automation of Software Deployment Processes: An Industrial Case Study. In *Proc. of IEEE SecDev 2018*. Cambridge, MA, USA.

[18] Vaishnavi Mohan and Lotf B. Othmane. 2016. SecDevOps: Is It a Marketing Buzzword? - Mapping Research on Security in DevOps. In *Proc. of ARES 2016*. Salzburg, Austria.

[19] David Sanán, Yongwang Zhao, Zhe Hou, Fuyuan Zhang, Alwen Tiu, and Yang Liu. 2017. CSimpl: A Rely-Guarantee-Based Framework for Verifying Concurrent Programs. In *Proc. of TACAS 2017*. Uppsala, Sweden.

[20] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous Deployment at Facebook and OANDA. In *Proc. of ICSE-C 2016*. Austin, TX, USA.

[21] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943.

[22] Philipp Stephanow and Niels Fallenbeck. 2015. Towards continuous certification of Infrastructure-as-a-Service using low-level metrics. In *Proc. of UIC 2015*. Beijing.

[23] Philipp Stephanow, Gaurav Srivastava, and Julian Schütte. 2016. Test-based cloud service certification of opportunistic providers. In *Proc. of IEEE CLOUD 2016*. San Francisco, CA, USA.

[24] Faheem Ullah, Adam Johannes Raft, Mojtaba Shahin, Mansooreh Zahedi, and Muhammad Ali Babar. 2017. Security Support in Continuous Deployment Pipeline. *CoRR* abs/1703.04277 (2017).

[25] Jamaiah Haji Yahaya, Aziz Deraman, Fauziah Baharom, and Abdul Razak Hamdan. 2009. Software certification from process and product perspectives. *IJCSNS* 9, 3 (2009), 222–231.