NFV-Bench: A Dependability Benchmark for Network Function Virtualization Systems

Domenico Cotroneo, Luigi De Simone, Roberto Natella

***Abstract*—Network Function Virtualization (NFV) envisions the use of cloud computing and virtualization technology to reduce costs and innovate network services. However, this paradigm shift poses the question whether NFV will be able to fulfill the strict performance and dependability objectives required by regulations and customers. Thus, we propose a *dependability benchmark* to support NFV providers at making informed decisions about which virtualization, management, and application-level solutions can achieve the best dependability. We define in detail the use cases, measures, and faults to be injected. Moreover, we present a benchmarking case study on two alternative, production-grade virtualization solutions, namely *VMware ESXi/vSphere* (hypervisor-based) and *Linux/Docker* (container-based), on which we deploy an NFV-oriented IMS system. Despite the promise of higher performance and manageability, our experiments suggest that the container-based configuration can be less dependable than the hypervisor-based one, and point out which faults NFV designers should address to improve dependability.**

***Index Terms*—NFV; Cloud Computing; Dependability; Fault Injection; Benchmarking; Hypervisors; Container-based Virtualization; Linux; Docker; VMware ESXi; VMware vSphere.**

I. Introduction

NETWORK Function Virtualization (NFV) is an emerging paradigm to enhance network equipment with the goal of reducing costs, and fostering competition and innovation [1]–[3]. NFV will take advantage of virtualization technologies to turn network functions and services (such as DPI, NAT, VoIP, etc.) into *Virtualized Network Functions* (VNFs), which will be implemented in software and executed as VMs on commodity hardware in high-performance data centers, namely *Network Function Virtualization Infrastructures* (NFVIs), and will possibly leverage cloud computing technologies.

Despite its recent introduction, NFV has already gained significant market traction [4]. A recent analysis [5] reports more than 100 products for VNFs, NFVIs, and MANO (*Management and Orchestration*) from more than 70 competing vendors. Moreover, in the foreseeable future, VNFs and NFVIs will be provided on a pay-per-use basis, according to the *as-a-service* business model [6].

In order to succeed, these NFV solutions have to compete not only in cost and manageability, but also in performance and reliability: telecom regulations impose *carrier-grade* requirements, in terms of extremely high packet processing rates and availability (99.999% or even higher) [7], [8]. However, the extensive use of *commercial off-the-shelf* (COTS) hardware and software components, which can be easily procured and replaced, exposes NFV systems to faults that must be automatically recovered within few seconds [8], [9].

These strict requirements impose accurate and fair procedures to *benchmark* the quality of service that can be obtained from competing NFV solutions. Benchmarks are an established practice for performance evaluation in the computer industry since decades. Examples of successful benchmarking initiatives are the TPC (Transaction Processing Performance Council [10]) and the SPEC (Standard Performance Evaluation Corporation [11]). More recently, the research community developed the notion of *dependability benchmarking* [12], [13], which evaluates the quality of service in the presence of faults, by using *fault injection*. This form of benchmarking has quickly matured from both the methodological and technical point of view, and has been applied on several domains such as transaction processing and automotive embedded systems [13]–[15]. The case for dependability benchmarking is especially compelling for NFV, as denoted by the interest of standardization bodies to define reliability requirements and evaluation procedures for the cloud and for NFV [8], [16].

In this work, we propose a dependability benchmark, namely *NFV-Bench*, to analyze faulty scenarios and to provide joint dependability and performance evaluations for NFV systems. This benchmark is aimed to support NFV service and infrastructure providers at making informed decisions about which NFVI, MANO and VNF solutions should be preferred.

To define *NFV-Bench*, we first review the prospective architecture of NFV systems, and discuss the potential use cases for dependability benchmarks in the context of NFV. Then, we define a set of *measures* both to evaluate the impact of faults on service level agreements (by quantifying the loss of the quality of service in the presence of faults), and to identify weaknesses in fault management mechanisms (by quantifying their coverage and latency). Moreover, we define a taxonomy of faults to be injected (*faultload*), by revisiting and unifying existing studies on faults in distributed computing systems.

We present an experimental case study based on the proposed dependability benchmark. We consider the case of an high-availability VoIP service using an NFV-oriented, open-source *IP Multimedia Subsystem* (IMS), namely *Clearwater* [17]. This NFV IMS has been deployed on two alternative NFVI configurations. In the first scenario, we consider an NFV stack composed by two commercial products (*VMware ESXi* and *VMware vSphere*); in the second scenario, the NFV stack uses two open-source technologies, namely *Linux* and *Docker*.

These scenarios represent two opposing virtualization paradigms: hypervisor-based and container-based virtualization. This dualism poses the question on which virtualization stack is more suitable for NFV. On the one hand, container-based virtualization is perceived as a promising solution to reduce the overhead of virtualization (as it does not emulate

This work has been supported by Huawei Technologies Co., Ltd., and by UniNA and Compagnia di San Paolo in the frame of Programma STAR (project FIDASTE).

The authors are with the Department of Electrical Engineering and Information Technology (DIETI) of the Federico II University of Naples, Italy, and with Critiware s.r.l., Italy. E-mail: {name.surname}@unina.it

a full computer machine), and thus to achieve better performance and scalability [18], [19]. On the other hand, traditional hypervisor-based virtualization enforces a stronger isolation among virtual machines and the physical machine, and it is regarded as a more secure and reliable solution [20]. We adopt the *NFV-Bench* benchmark to analyze the trade-offs of these solutions with respect to performance and dependability, which are both important concerns in the context of NFV. The experimental results point out that:

- Achieving high-availability is not simply a matter of quickly restarting VMs or containers. Even if containers can achieve a quicker restart time, there are other aspects of fault management that have a stronger impact on the availability of the NFV system. We found that, when the state of the hypervisor or VMs becomes unstable (e.g., due to internal errors), ESXi forces the shutdown of the host to trigger a failover on another machine. Instead, Linux/Docker attempts to ignore errors and continue execution, thus hindering the fault recovery process.
- Moreover, we found that the hypervisor-based configuration is more effective to protect VNFs from memory contention (e.g., due to a buggy/overloaded VNF on a shared host), by using memory reservations, a more efficient memory allocator, and reporting diagnostic information. Instead, Linux/Docker needs additional monitoring tools to manage these cases.
- The latency of the recovery process is quite large, and the NFV designers need to tune fault management (such as heartbeat mechanisms in VMware) to meet the requirements of NFV. Fault injection can be used to evaluate and validate the latency of fault detection and recovery.
- The robustness of fault management components is itself a concern. We found that network and storage data corruptions could not be recovered, due to residual errors caused by the corruptions. In general, the dependability benchmark allows to understand the limitations of fault management products, which are often not well documented or validated by vendors.

The paper is organized as follows. Section II provides background about dependability benchmarking and fault injection. Section III presents our dependability benchmark for NFV systems, and discusses in detail the use cases, measures and faultload. In section IV, we introduce the NFV IMS case study. In section V we provide the experimental results, and in section VI we discuss them and conclude the paper.

II. Background and Related Work

Basic concepts on dependability benchmarking. The goal of dependability benchmarking is to quantify the dependability properties of a computer system or component, in a fair and trustworthy way [13], [21]. This goal is especially important for COTS-based systems, since it enables system designers to make informed purchase and design decisions. The research efforts in this area culminated with the definition of a general framework for dependability benchmarking by the *DBench* project [12], [13]. In this framework, the benchmark precisely defines the measures, procedures and conditions, in order to guide the adoption by its stakeholders (including product vendors, system integrators and providers, and the users). More recently, these general concepts were integrated in the *ISO/IEC Systems and software Quality Requirements and Evaluation* (SQuaRE) standard [22], which defines an evaluation module (*ISO/IEC 25045*) for "recoverability", which is defined as the ability of a product to recover affected data and re-establish the state of the system in the event of a failure.

A dependability benchmark distinguishes between the **Benchmark Target** (BT), which is the component or system to be evaluated, and the **System Under Benchmark** (SUB), which includes the BT along with other resources (both hardware and software) needed to run the BT. The evaluation process is driven by the **benchmark context**, which includes the *benchmark user* (which takes advantage of the results) and the *benchmark performer* (which carries out the experiments), and the *benchmark purpose* (such as, the evaluation of fault-tolerance features supported or claimed by a product, or the integration and interoperability testing of a larger system). Finally, the benchmark defines **measures** for the dependability and performance of the BT, either qualitative (e.g., supported fault-tolerance capabilities) or quantitative (e.g., error rate and response time in the presence of faults).

Dependability benchmarks apply two forms of stimuli on the system, namely the **workload** and the **faultload**. The workload represents the typical usage profile for the SUB, and it is defined according to workload characterization techniques that are used for classical performance benchmarks [23]. For example, dependability benchmarks on DBMSes and web servers extended the workloads from *TPC-C* [10] and *SPECweb* [11]. The faultload is a peculiarity of dependability benchmarks: it defines a set of exceptional conditions that are injected, to emulate the ones that the system will experience in operation. The definition of a realistic faultload is the most difficult part of defining a dependability benchmark [12]. The most important source of information is the post-mortem analysis of failure data in operational systems, which can be gathered from empirical studies, or from end-users and providers. Alternatively, the faultload can be identified from a systematic analysis of system's components and their potential faults, based on expert judgment. A common approach is to define selective faultloads, each addressing different categories: hardware, software, and operator faults [14], [24]–[28].

It is important to remark that dependability benchmarks separate the BT from the so-called **Fault Injection Target** (FIT), that is, the component of the SUB subject to the injection of faults. This separation is important since it is desirable not to modify the BT when applying the faultload, in order to get the benchmark accepted by its stakeholders. For this reason, fault injection introduces perturbations outside the BT. Moreover, this approach allows to compare several BTs with respect to the same set of faults, since the injected faults are independent from the specific BT.

Once these elements are defined, a dependability benchmark entails the execution of a sequence of fault injection experiments. In each experiment, the SUB is first deployed and configured; then, the workload is submitted to the SUB and, during its execution, faults from the faultload are injected; at the end of the execution, performance and failure data are collected from the SUB, and the testbed is cleaned-up before starting the next experiment. This process is repeated several times, by injecting a different fault while using the same workload and collecting the same performance and failure data. The execution of fault injection experiments is typically

supported by tools for test automation, workload generation, and fault injection [29]–[31].

Fault injection can also be leveraged to evaluate *performability*, which joins performance and availability into a combined measure. Performability is typically computed through stochastic modeling, such as using Reward Markov Models [32], which assign a numeric score to each state to reflect the quality of service under that state (e.g., < 1 for a degraded state, in which one of the replicas of a component is failed). Fault injection is complementary to these approaches, as it provides parameters for stochastic models, such as the *coverage* and *latency* [33]–[35].

State-of-the-art and open issues. The general principles of dependability benchmarking, as defined by DBench [12], [13] and the ISO/IEC 25045 [22], have been specialized for several different domains. The most well-known benchmarks have been aimed at OSes (including UNIX and Windows [36]–[38]), DBMSes [14], web servers [26], and embedded systems [15]. These studies defined measures to evaluate their specific target systems, such as throughput, response time, error rate, and availability for DBMSes and web servers, and reboot time and failure severity for OSes (e.g., process VS kernel failures); and specific faultloads, such as invalid configurations for DBMSes and webservers, and invalid system calls and device driver bugs for OSes.

Cloud services and infrastructures, such as NFV systems, still lack mature dependability benchmarks. On the one hand, the studies on dependability benchmarking focused on other domains, or generically on traditional IT infrastructures. However, these proposals did not take into account the peculiarities of NFV and of cloud computing, and in particular the *as-a-service* model, which involves new stakeholders and use cases (thus requiring to rethink the roles of the benchmark performers and users, and of benchmark elements) and different fault types. Moreover, differing from traditional IT infrastructures, NFV has more stringent performance and dependability requirements inherited from telecom applications, which require new benchmark measures to account for the QoS under faults. We address these aspects in this paper, as discussed in § III.

On the other hand, the state-of-the-art of fault injection in cloud computing is fragmented in many different tools that only address specific issues of cloud computing and virtualization software. Well-known solutions in this field include *Fate* [39] and its successor *PreFail* [40] for testing cloud-oriented software (such as Cassandra, ZooKeeper, and HDFS) against faults from the environment, by emulating at API level the unavailability of network, storage, and remote processes; similarly, Ju *et al.* [41] and *ChaosMonkey* [42] test the resilience of cloud infrastructures by injecting crashes (e.g., by killing VMs or service processes), network partitions (by disabling communication between two subnets), and network traffic latency and losses; *CloudVal* [43] and Cerveira et al. [44] tested the isolation among hypervisors and VMs by emulating hardware-induced CPU and memory corruptions, and resource leaks (e.g., induced by misbehaving guests). In summary, these studies focus fault injection on testing specific parts of the cloud stack with respect to specific classes of faults (e.g., network or CPU faults).

We remark that dependability benchmarking goes beyond the testing of individual components. In fact, the dependability of NFV systems results from tight interactions among several components, where fault-tolerance mechanisms are introduced at several layers (e.g., at application and at MANO level, as discussed in § III). Therefore, we propose a methodology to jointly evaluate performance and dependability from the perspective of NFV systems as a whole, including both service-level measures (as in our previous work [45]), and infrastructure-level measures for evaluating fault management aspects. Moreover, dependability benchmarking gives emphasis to the representativeness and portability of the faultload, in order to support a fair comparison: thus, we introduce a fault model that spans the whole NFV stack (including both the virtualization and physical layers) and that unifies the existing fault classes in a common framework, not limited to specific cloud technologies or to specific classes of faults.

III. Dependability Benchmarking in NFV

To define a dependability benchmark for NFV, we first analyze the benchmark elements according to its stakeholders and use cases (§ III-A). Then, we address the problem of defining appropriate benchmark measures (§ III-B), faultload (§ III-C) and workload (§ III-D) for NFV.

A. Benchmark elements

In the ETSI NFV framework [1], the architecture of a virtualized network service can be decomposed in three layers, namely the *service*, *virtualization*, and *physical* layers (Fig. 1). In the service layer, each VNF provides a traffic processing capability, and several VNFs are combined into a *service function chain* (SFC) to provide added-value network services. Examples are traffic shaping, billing, and deep packet inspection functions that can be introduced as gateways to provide security and quality of service. The topology of interconnected VNFs is represented by a *VNF forwarding graph* [1], which is transparently deployed and managed by the NFVI.

VNFs are implemented in software, and are executed within VMs and networks deployed on the physical resources of the NFVI. For example, in Fig. 1 each VNF is mapped to a pool of VM replicas, with an additional VM to perform load balancing, and a virtual network segment to connect them. These VMs are scaled out and dynamically mapped to physical machines (PMs) to achieve high performance and resource efficiency. These operations are overseen by MANO software, that deploys and controls the VNFs by interacting with a Virtualized Infrastructure Manager (VIM) component inside the NFVI. Moreover, the MANO orchestrates fault management, by correlating data from the NFVI and the VNFs, and by reconfiguring the network in the case of faults.

To identify the benchmark elements (SUB, BT, FIT), we need to consider the potential use cases of the dependability benchmark. Such use cases involve, as users and performers of the benchmark, *telecom service providers* and *customers*, *NFV software vendors*, and *NFV infrastructure providers*.

▹ ***Case #1***. A telecom service provider designs a network service by composing a VNF service chain, using VNF software developed in-house or provided by third-party NFV software vendors. The telecom provider performs the benchmark to get confidence that the network service as a whole is able to achieve QoS objectives even in worst-case (faulty) conditions. End-users and other telecom providers, which consume network services on a pay-per-use basis (*VNFaaS*), are the

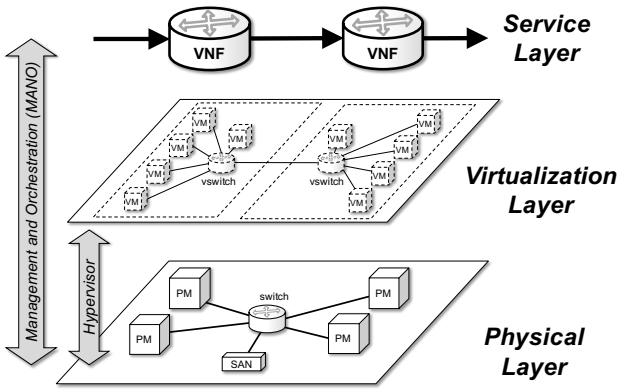


Fig. 1: Architectural layers in NFV.

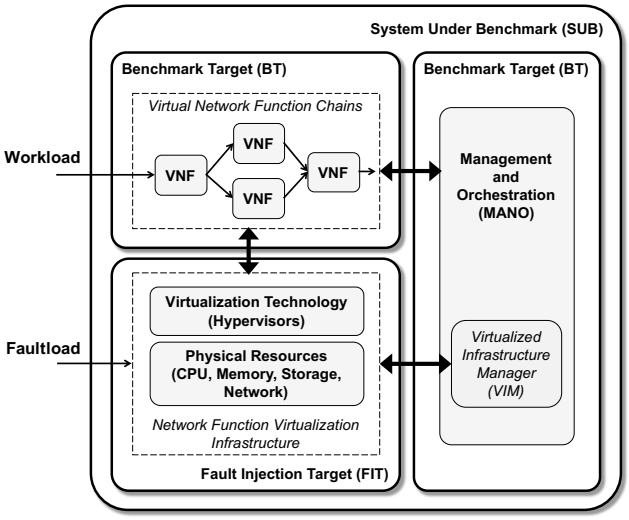


Fig. 2: Elements of the dependability benchmark.

benchmark users: they can demand empirical evidence of high dependability and performance, to make informed decisions based on both cost and quality concerns. The telecom provider can produce this evidence in cooperation with an independent certification authority, in a similar way to certification schemes for cloud services [16], [46], [47].

▹ ***Case #2***. An NFV infrastructure provider setups an environment to host VNFs on a pay-per-use basis (*NFVIaaS*) for telecom operators, which are the benchmark users. The NFVI is built by acquiring off-the-shelf, industry-standard hardware and virtualization technologies, and by operating them through MANO software provided by NFV software vendors. The NFVI provider performs the dependability benchmark to get confidence on the dependability of its configuration and management policies, with respect to faults of hardware and virtualization components; and revises the configuration and the MANO according to the feedback of the benchmark.

In both use cases, the SUB must include the service layer, the infrastructure level (both virtual and physical), and the MANO. These elements are showed in Fig. 2. The BT is represented, respectively, by the VNF service chain, and by the MANO. In the former use case, the benchmark provides feedback on the robustness of VNF software and of the service chain. In the latter, the benchmark provides feedback on fault management mechanisms and policies of the MANO. In both cases, the FIT is represented by the NFVI (both physical and virtualization layers): the NFVI is built from COTS hardware and virtualization components that are relatively less reliable than traditional telecom equipment, and thus represent a dependability threat for the NFV system. Thus, the benchmarking process injects faults in the NFVI, in order to assess the VNF service QoS in spite of faults in the NFVI, and to evaluate and tune MANO software.

B. Benchmark measures

The measures are a core part of a dependability benchmark, since they represent the main feedback provided to the benchmark users. According to the previous two use cases, we identify two groups of benchmark measures:

- *Service-level* (VNF) measures, which characterize the quality of service as it is perceived by VNF users, in terms of performance and availability of VNF services;
- *Infrastructure-level* (NFVI) measures, which characterize the NFVI in terms of its ability to detect and to handle faulty conditions caused by hardware and software components inside the infrastructure.

In practice, infrastructure-level components influence the service-level measures, since the availability and performance perceived by VNF users will be better if the infrastructure is quicker at detecting and handling faults, as more resources will be available for applications. Thus, the two types of measures are meant to provide complementary perspectives to the benchmark stakeholders. The service-level measures are aimed at telecom service providers and customers, and should be interpreted as a comprehensive evaluation of the NFV system as a whole, inclusive of the indirect effects of fault-tolerance mechanisms on service availability and performance. Instead, the infrastructure-level measures provide more detailed insights on fault detection and recovery actions inside the NFV system, regardless of their impact on the quality of service, and are useful for infrastructure providers to understand and improve the fault management process.

1) Service-level measures: We define service-level measures to connect the results of the dependability benchmark to SLA (Service Level Agreement) requirements of the VNF services. Typically, SLA requirements impose constraints on service performance in terms of latency and throughput, and on service availability in terms of outage duration.

The service-level measures of the benchmark include the *VNF latency* and the *VNF throughput*. It is important to note that, while latency and throughput are widely adopted for performance characterization, we specifically evaluate latency and throughput *in the presence of faults* in the underlying NFVI. We introduce these measures to quantify the impact of faults on performance, and evaluate whether the impact is small enough to be acceptable. In fact, it can be expected that performance will degrade in the presence of faults, leading to higher latency and/or lower throughput, since less resources will be available (due to the failure of components in the NFVI) until fault management completes.

▹ ***VNF Latency***. In general terms, network latency is the delay that a message "takes to travel from one end of a network to

another" [48]. A similar notion can also be applied to network traffic processed by a VNF, or, more generally, by a chain of interconnected VNFs, as in Fig. 1. Latency can be evaluated by measuring the time between the arrival of a unit of traffic (such as a packet or a service request) at the boundary of the chain, and the time at which the processing of that unit of traffic is completed (e.g., a packet is routed after inspection; or a response is provided to the source of a request).

The *VNF latency* is represented by percentiles of the empirical cumulative distribution function (CDF) of traffic processing times, and denoted by $L(x) = P(l < x)$, where l is the latency between the request and the response for a network traffic unit. We compute the CDF by considering all traffic units sent after that a fault has been injected in the NFV system ($t^{\texttt{injection}}$), and until the end of the execution of the experiment ($t^{\texttt{end}}$). Thus, the CDF denotes the ability of the NFV system to provide high performance despite the occurrence of a fault. For evaluation purposes, SLAs typically consider the 50th and the 90th percentiles of the CDF (i.e., $L(50)$ and $L(90)$) to characterize the average and the worst-case performance of telecommunication systems [49].

▹ ***VNF Throughput***. In general, throughput is the rate of successful operations completed within an observation interval. In our context, the *VNF throughput* is represented by the rate of processed traffic (e.g., packets or requests per second) by VNF services in presence of faults. The VNF throughput of an experiment is given by $T = \frac{N}{t^{\texttt{end}} - t^{\texttt{injection}}}$, where N is the total number of traffic units between the injection and the end of the experiment.

The dependability benchmark aggregates VNF latency percentiles and throughput values from several fault injection experiments. Different experiments can inject different types of faults (as discussed later in section III-C) in different parts of the NFV system (i.e., at each experiment, the same fault type can be injected on different instances of a resource). From these experiments, the overall performance of the NFV system can be summarized by statistics such as the *maximum and minimum value among latency percentiles*, and the *average value among throughput values*, which quantify the extent of performance degradation under faults. These aggregated values can be computed over the entire set of fault injection experiments to get an overall evaluation of NFV systems. Another approach is to divide the set of experiments into subsets, with respect to the injected fault type, or with respect to the component targeted by fault injection, and then to compute aggregate values for each subset. Among these experiments, benchmark users are interested to know in which ones the latency percentiles and throughput *exceeded their reference values*, which point out the specific faults or parts that expose VNFs to performance issues.

Finally, we introduce the *VNF unavailability* to evaluate the ability of VNFs to avoid, or to quickly recover from service outages. Differing from latency and throughput, which assess whether faults cause performance degradation, this measure evaluates whether faults escalate into user-perceived outages. SLAs require that service requests must succeed with a high probability, which is typically expressed in *nines*. It is not unusual that telecom services must achieve an availability not lower than 99.999%, i.e., the monthly "unavailability budget" amounts to few tens of seconds per month [49]. For example, the ETSI "NFV Resiliency Requirements" [8] report that voice call users and real-time interactive services do not tolerate more than 6 seconds of service interruption, while the TL-9000 forum [7] has specified a service interruption limit of 15 seconds for traditional telecom services. Service disruptions longer than a few tens of seconds are likely to worsen the user experience (as they impact not only on isolated service requests, but also on the user retries [49]), thus accruing the perceived downtime. Therefore, we introduce a benchmark measure to evaluate the duration of VNF service outages.

▹ ***VNF Unavailability***. A VNF service is considered unavailable if either traffic is not processed within a maximum time limit (i.e., it is timed-out), or errors are signaled to the user. The *VNF Unavailability* is defined as the amount of time during which VNF users experience this behavior.

During an experiment, after the injection of a fault, the VNF service may become unavailable (i.e., the rate of service errors exceeds a reference limit), and return available when the service is recovered. Moreover, a service may oscillate from availabile to unavailable, and viceversa, for several times during an experiment (e.g., there are residual effects of the fault that cause sporadic errors). Thus, the VNF Unavailability is given by the sum of the "unavailability periods" (denoted with i) occurred during an experiment:

$$U = \sum_i t_i^{\texttt{avail}} - t_i^{\texttt{unavail}}$$

where:

$$t^{\texttt{injection}} < t_i^{\texttt{unavail}} < t_i^{\texttt{avail}} < t^{\texttt{end}}, \ \forall i$$

$$\text{error-rate}(t) > \text{error-rate}^{\texttt{reference}}, \forall t \in [t_i^{\texttt{unavail}}, t_i^{\texttt{avail}}], \forall i.$$

If the VNF service does not experience any user-perceived failure, the VNF Unavailability is assumed to be $U = 0$. If the VNF service is unable to recover within the duration of the experiment (say, ten minutes), we conclude that the VNF cannot automatically recover from the fault, and that it needs to be manually repaired by a human operator. In this case, recovery takes orders of magnitude more time than required by SLAs: we mark this case by assigning $U = \infty$.

The dependability benchmark aggregates VNF Unavailability values from fault injection experiments, by identifying the *experiments in which VNFs cannot be automatically recovered* (i.e., VNF Unavailability is a finite value), which points out the scenarios that need to be addressed by the NFV system designers. Moreover, the *maximum* and the *average* of *(finite) VNF Unavailability values* indicate how much the VNF is able to mask the faults. In a similar way to performance measures, the aggregated values of VNF Unavailability can be computed over subsets of fault injection experiments (e.g., divided by type or target of injected faults) for a more detailed analysis.

2) Infrastructure-level measures: Infrastructure-level measures are aimed at providing insights to NFVI providers on fault-tolerance algorithms and mechanisms (FTAMs) of their infrastructure [33]. Several fault-tolerance strategies are available to NFV system designers, and are discussed in the ETSI document on "NFVI Resiliency Requirements" [8]. These FTAMs are provided at hypervisor and MANO level, and VNF services should be designed and configured to take advantage of them. FTAMs are broadly grouped in two areas:

- **Fault Detection**: FTAMs that notice a faulty state of a component (such as a VM or a physical device) as

soon as a fault occurs, to timely start the fault recovery process. Examples of fault detection are: heartbeats and watchdogs, which periodically poll the responsiveness of a service or of a component; performance and resource consumption monitors; internal checks performed internally in each component (such as the hypervisor) to report on anomalous states, such as failed I/O operations and resource allocations; data integrity checks.

- **Fault Recovery**: FTAMs that perform actions to counteract a faulty component. An example of recovery action for NFVIs include the (de-)activation of VM instances and their migration to different hosts, or retrying a failed operation. Moreover, VMs and physical hosts can be reconfigured, e.g., by updating a virtual network configuration or deactivating a faulty network interface card.

Since implementing all these solutions in the fault management process can be a complex task, we introduce benchmark measures to quantitatively evaluate their effectiveness. These measures are complementary to service-level measures, as they provide feedback to NFVI providers about individual FTAMs (heartbeats, watchdogs, logs, etc.).

▹ ***Fault Detection Coverage and Latency***. We define the *Fault Detection Coverage* (FDC) as the *percentage of fault injection tests in which the NFV system issues a fault notification*, either on individual nodes of the infrastructure (e.g., hypervisor logs), or on MANO software. A recovery action can only be triggered after that a fault has been detected. The FDC is computed by counting both the number of tests in which the injected fault is reported by the NFV system ($\#F_{\texttt{fault\_detected}}$), and tests in which the injected fault is not reported but causes service failures or performance degradations ($\#F_{\texttt{fault\_undetected}}$):

$$FDC = \frac{\#F_{\texttt{fault\_detected}}}{\#F_{\texttt{fault\_undetected}} + \#F_{\texttt{fault\_detected}}} .$$

The *Fault Detection Latency* (FDL) is the time between the injection of a fault ($t^{injection}$), and the occurrence of the first fault notification ($t^{detection}$). The Fault Detection Latency is computed for the subset of experiments in which a fault has actually been detected. The FDL is given by:

$$FDL = t^{detection} - t^{injection}$$

▹ ***Fault Recovery Coverage and Latency***. The *Fault Recovery Coverage* (FRC) is the percentage of tests in which a recovery action (triggered by fault detection) is successfully completed. For example, in the case of a VM restart, the recovery is considered successful if a new VM is allocated, and VNF software is correctly started and executed. The FRC is represented by the ratio between the number of tests in which faults were detected ($\#F_{\texttt{fault\_detected}}$), and the number of tests in which the recovery action is *successfully completed* ($\#F_{\texttt{fault\_recovered}}$):

$$FRC = \frac{\#F_{\texttt{fault\_recovered}}}{\#F_{\texttt{fault\_detected}}} .$$

For those experiments in which the NFV system is able to perform a recovery action, we define the *Fault Recovery Latency* (FRL) as:

$$FRL = t^{recovered} - t^{detected}$$

where $t^{recovered}$ denotes the time when the NFV system concludes a recovery action.

C. Faultload

The faultload is the set of faults and exceptional conditions that are injected to evaluate dependability properties [12]. The definition of a "good" faultload is a tricky task, since the fault model needs to be *complete* and *representative* with respect to real faults that the system will experience during operation [12], [13]. Moreover, the faultload should be *generic* enough to be applicable to different NFV systems (e.g., it should be independent from specific virtualization and hardware products). Finally, the design of the faultload should be supplied with practical indications on how these faults should be emulated in the context of a concrete system.

We follow a systematic approach to address these properties in the faultload. To cover the *complete* architecture of an NFV system [50], and to be *generic* enough to apply faults on different NFV implementations, we define a fault model for the four domains of the NFV architecture: *network*, *storage*, *CPU*, and *memory*. These elements are present both as *virtual* resource abstractions, and as *physical* resources of the NFVI (Fig. 1). The benchmark performer can use the fault model for defining the faultload for the target NFVI, by first enumerating the resources in the NFVI, and then by systematically applying the fault model on each resource. The fault model includes *physical and virtual CPU and memory faults*, to be applied on each physical and virtual machine in the NFVI. Moreover, the fault model includes *physical and virtual storage and network faults*, to be applied on each virtual and physical storage interface, disk, network interface, or switch in the NFVI.

For each domain of NFV (CPU, memory, disk and network), the fault model defines three categories of faults: *unavailability* (the resource becomes inaccessible or unresponsive); *delay* (the resource is overcommitted and slowed down); *corruption* (the stored or processed information is incorrect). These categories broadly include all the possible faulty states of a component, and are inclusive of failure categories defined in previous studies [21], [51], [52]. We specialize these general fault categories, and get a set of *representative* faults, by analyzing which hardware, software, and human faults are likely to occur for each category and for each domain [12], [13], by revisiting the scientific literature on fault injection and failure analysis in cloud computing infrastructures [39]–[44], [53], [54], well-known cloud computing incidents [55], [56], and the analysis of the prospective architecture and products for NFVIs [1], [5].

The fault mode has been summarized in TABLE I, which shows, for each domain and for each fault type, the possible root causes of the faults (from software, hardware, and operators), and the fault effects to be injected. The analysis identified the following fault types.

Physical CPUs and memory: These physical resources can become abruptly broken due to wear-out and electrical issues. If these faults are detected by machine checks, they lead to CPU exceptions and to the de-activation of failed CPU and memory banks. Otherwise, these faults cause silent corruptions of random bytes in CPU registers and memory banks; even in the case of ECC memory, data can become corrupted before it is stored in memory (e.g., when flowing through

TABLE I: Overview of the fault model.

		Root cause	Fault Model
Physical CPU	*Unavailability*	Physical CPU permanently broken [hw.]	Physical CPU disabled
	Delay	Physical machine is overloaded [op., sw.]	CPU hog in the virtualization layer context
	Corruption	Electromagn. interferences, virtualization layer bugs [hw.]	CPU register corruption in the virtualization layer context
Virtual CPU	*Unavailability*	Insufficient capacity planning [op.]	Virtual CPU disabled
	Delay	Virtual machine is overloaded [op., sw.]	CPU hog in VM context
	Corruption	EMIs, virtualization layer bugs [sw., hw.]	CPU register corruption in VM context
Physical memory	*Unavailability*	Physical memory bank permanently broken [hw.]	Memory hog in the virtualization layer context
	Delay	Virtual machine is overloaded [op., sw.]	Memory thrashing in the virtualization layer context
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.]	Memory page corruption in the virtualization layer context
Virtual memory	*Unavailability*	Insufficient capacity planning [op.]	Memory hog in the virtual node context
	Delay	Virtual machine is overloaded [op., sw.]	Memory thrashing in the virtual node context
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.]	Memory page corruption in virtual node context
Physical network	*Unavailability*	NIC or network cable permanently broken [hw.]	Physical NIC interface disabled
	Delay	Network link saturated [op.]	Network frames delayed on physical NIC
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.]	Network frames corrupted on physical NIC
Virtual network	*Unavailability*	Misconfiguration [op.]	Virtual NIC interface disabled
	Delay	The virtualization layer is overloaded [op., sw.]	Network frames delayed on virtual NIC
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.]	Network frames corrupted on virtual NIC
Physical storage	*Unavailability*	HBA or storage cable permanently broken [hw.]	Physical HBA interface disabled
	Delay	Storage link saturated [op.]	Physical storage I/O delayed
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.]	Physical storage I/O corrupted
Virtual storage	*Unavailability*	Misconfiguration [op.]	Virtual HBA interface disabled
	Delay	The virtualization layer is overloaded [op., sw.]	Virtual storage delayed
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.]	Virtual storage corrupted

the CPU or the bus). Software faults in the VMM (Virtual Machine Monitor) may cause the corruption of entire memory pages, due to buggy memory management mechanisms (such as page sharing and compression) or to generic memory management bugs at VMM level (such as buffer overruns and race conditions). Finally, physical CPUs and memory can be overloaded by an excessive number of VMs, or by buggy services running in the VMM; in turn, CPU and memory contention leads to scheduling and allocation delays.

Virtual CPUs and memory: The virtual CPUs and virtual memory of VMs may not be allocated due to insufficient resources reserved for a VM. Moreover, software and operator faults inside the VM may overload virtual CPUs and memory. Finally, in a similar way to physical CPUs and memory, electrical issues and VMM bugs may lead to data corruptions, but in the context of VM (e.g., corrupting the state of the guest OS or VNF software).

Physical storage and network: Storage and network links (respectively, HBA and NIC interfaces, and connections between machines, and network switches and storage) may fail due to wear-out and electrical issues. Moreover, electrical issues and software bugs in device drivers may cause the corruption of block I/O and network frames. The storage and network bandwidth may get saturated due to excessive load and insufficient capacity, causing I/O delays.

Virtual storage and network: Storage and network interfaces of individual VMs, and virtual switch and storage connections, may become unavailable due to human faults in their configuration. In a similar way to physical storage and network, wear-out and electrical issues may affect the I/O traffic of specific VMs, and the I/O traffic may be delayed due to bottlenecks caused by the emulation of virtual switches and storage.

These faults are feasible to implement using established fault injection techniques [31]. Corruption faults can be implemented with SWIFI techniques (*Software-Implemented Fault Injection*), which emulate hardware and software faults by injecting the expected *effects* produced by these faults on the target software [33]. This approach is practically convenient since it avoids to simulate the actual root cause (such as, a broken CPU or electromagnetic interference), which would entail excessive costs and efforts. For example, simulations and empirical studies [57]–[61] showed that physical CPU faults can accurately be emulated by corrupting the state of CPU registers through *bit-flipping*. A similar approach can be applied to unavailability and delay faults, by saturating a resource with artificial load, by turning it off, or by intercepting API calls (e.g., on the device drivers' interface) and forcing their failure. We developed a custom fault injection tool suite that applies these techniques in Linux and VMware ESXi. The tool suite provides a set of modules to be loaded in the hypervisor or guest OS, which modify the state of resources at run-time; implementation details are described in [62].

The benchmark users and performers can selectively use only part of the faultload, if they need to focus the benchmark according business priorities, or want to leverage their knowledge of faults that are more likely in their specific NFV system. For example, software faults in the VMM (in particular, bugs in device drivers and in other VMM extensions developed by untrusted third-parties) may not be injected, depending on the integrity and maturity of a VMM product. Overloads may be omitted in the case that resources are overprovisioned, or in the case that capacity planning has been carefully performed. Human operator faults may be omitted in the case that configuration of the virtualization layer is fully automated, or that configuration policies are carefully checked. Moreover, benchmark users may give different weights to faults when aggregating the results from different faults [45].

D. Workload

The definition of the workload specifies how to exercise the NFV system while the faultload is injected. In order to obtain reasonable and realistic results from dependability benchmarks, these workloads should be representative of the workload that the NVF will face once deployed. Typical workloads for testing networks follow pre-defined patterns or statistical distributions [63]. A realistic workload for NFV systems can be automatically generated using load generators for performance benchmarking. Note that the selection of workloads also depends on the kind of network service that is hosted on the NFVI. For this reason, we refer the reader to

existing network performance benchmarks and network load generators. Relevant examples of workloads for NFVIs are represented by performance benchmarks designed for cloud computing systems [64]–[66] and by network load testing tools such as Netperf.

IV. Case Study

To better understand how to apply the dependability benchmark, and to showcase the results that can be obtained from it, we perform an experimental analysis on an NFV system running a virtualized *IP Multimedia Subsystem* (IMS). We deploy the IMS on two NFVIs based on different virtualization technologies: a commercial hypervisor-based virtualization platform (*VMware ESXi*), and an open-source container-based solution (*Linux* containers). The ETSI envisions the use of both hypervisor-based and container-based virtualization for NFV [67], and these two products are going to be extensively used in NFVIs [68], [69]. Moreover, we consider two virtualization management solutions, *VMware vSphere* and *Docker*, paired respectively with VMware ESXi and Linux.

A. The Clearwater IMS

The VNF software comes from the *Clearwater* project [17], which is an open-source implementation of an IMS for cloud computing platforms. Its main components are:

- *Bono*: the SIP edge proxy, which provides both SIP IMS Gm and WebRTC interfaces to clients. The Bono node is the first point of contact between clients and the IMS. Clients are linked to a specific Bono node at registration.
- *Sprout*: the SIP registrar and authoritative routing proxy, which handles client authentication, and provides the bulk of I-CSCF and S-CSCF functions.
- *Homestead*: handles authentication credentials and user profiles, using a Cassandra datastore [70].
- *Homer*: XML Document Management Server that stores MMTEL service settings, using a Cassandra datastore.
- *Ralf*: a component that provides billing services.

B. Testbed

Figure 3 shows the testbed used for the experimental evaluation of the IMS NFV system. The same testbed has been used for experiments with both virtualization technologies, by switching between VMware ESXi with vSphere, and Linux containers with Docker, and using the same IMS VNF software in both configurations. The testbed includes:

- *Host 1*: A machine equipped with an Intel Xeon 4-core 3.70GHz CPU and 16 GB of RAM. In the hypervisor-based scenario, this machine runs the VMware ESXi hypervisor v6.0, and hosts VMs running the VNFs of Clearwater (one VM for each VNF). In the container-based scenario, this machine runs Ubuntu Linux 14.04 OS and Docker v1.12, and hosts containers running the VNFs (one container for each VNF). Faults are injected in the resources (virtual and physical) of this host.
- *Host 2*: A machine with the same hardware and software configuration of Host 1. Moreover, this machine hosts active replicas of the same VNFs of Host 1, to provide redundancy to tolerate faults in the other host.

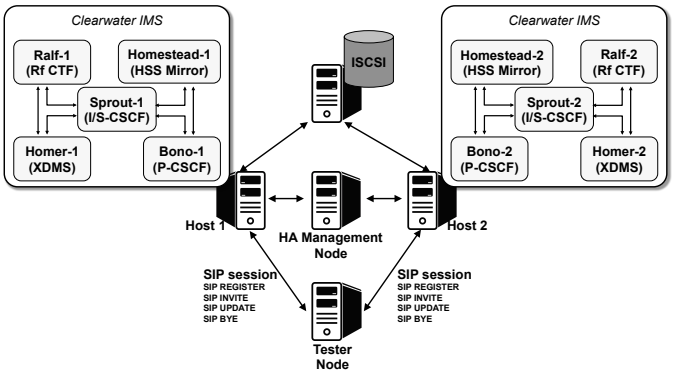


Fig. 3: The IMS NFV testbed.

- *Name, Time and Storage Server*: A machine that hosts network services (DNS, NTP) to support the execution of VNFs. Moreover, this machine hosts a shared storage service with iSCSI. The shared storage holds the persistent data of Cassandra managed by the *Homestead* and *Homer* VNFs, and the virtual disk images of the VNFs.
- *High-Availability (HA) Management Node*: A machine that runs management software: in the hypervisor-based configuration, this machine runs the VMware HA service of VMware vSphere; in the container-based configuration, this machine runs the Docker Swarm master node.
- *Tester Host*: A Linux-based computer that runs an IMS workload generator. Moreover, this machine runs a set of tools for managing the experimental workflow. These tools interact with the NFVI to deploy the VNFs, to control fault injection tools installed on the target Host 1, and to collect performance and failure data both from the workload generator and from the nodes of the NFVI.
- *Load Balancer*: A machine that forwards and balances IMS requests to Bono VNFs on the two host machines.
- A Gigabit Ethernet LAN connecting all the machines.

The workload set-ups several SIP sessions (calls) between end-users of the IMS. Each SIP session includes a sequence of requests for registering, inviting other users, updating the session, and terminating the session. This workload is generated by *SIPp* [71], an open-source tool for load testing of SIP systems. An experiment exercises the IMS by simulating 10,000 users and 10,000 calls.

We consider a high-availability configuration, where each VNF is replicated across the hosts (Fig. 3). The Clearwater VNFs are designed to be stateless and horizontally scalable, and to balance SIP messages between replicas with round-robin DNS. We also enable fault-tolerance capabilities provided by MANO software. In VMware vSphere, the *HA cluster* [72] capability automatically restarts a VM (in the case of VM failures) or migrates it on another node (in the case of physical host failures). In the Docker configuration, we enabled *Docker Swarm* [73] to provide failure detection and automated restart of containers. To allow migration in VMware HA, we stored all VNF data (including the VM disk image) on the shared iSCSI storage; in Docker, the iSCSI storage is used to store the *Cassandra* datastores of *Homer* and *Homestead*.

Faults are injected in the Host 1 and its VNF replicas. We focus on injecting faults in the physical host and in *Homestead* and *Sprout*, as these are the two main VNFs strictly required

by the use cases of the IMS. Each experiment lasts for 300*s*. We inject a fault after 80*s* from the start the workload, and remove the fault after 60*s*. As discussed in § III-C, we consider both *I/O* and *CPU/memory* faults. Network and storage corruptions, drops, and delays are injected in the network and storage interfaces of the host, and on the virtual network and storage interfaces of *Homestead* and *Sprout*. We performed five repeated experiments for each type of fault. Overall, for each testbed configuration (hypervisor- and container-based), we perform 180 experiments (60 on the physical layer, 120 on the virtual layer), for a total of 360 experiments.

V. Experimental results

We computed the measures defined by our dependability benchmark using experimental data on performance and failures. As basis for comparison, we computed the latency and throughput of the IMS NFV system in fault-free conditions, and compare them to the measures in faulty conditions to quantify the performance loss of the IMS. Moreover, as basis to compare the IMS unavailability, we consider that the service cannot be unavailable for more 30 seconds. This choice is an optimistic bound for the unavailability of network services, as the budget can be even stricter for highly-critical services; but, as we will see in our analysis, achieving this goal using IT virtualization technologies is still a challenging problem.

In the following, we divide the analysis in three parts: service-level evaluation with fault injection at the physical layer; service-level evaluation with fault injection at the virtualization layer; and infrastructure-level evaluation.

A. Service-level evaluation, faults in the physical layer

Fig. 4, Fig. 5, and Fig. 6 show respectively the *VNF unavailability*, *VNF throughput* (for both SIP REGISTERs and INVITEs) and *VNF latency* (for SIP REGISTERs) computed from fault injection experiments, for both the ESXi/vSphere and the Linux/Docker testbed. On the *x*-axis, the plots show the fault types of the fault model (§ III-C).

The experiments pointed out that physical faults have a noticeable impact in terms of unavailability of the NFV system. The extent of the unavailability varies across the fault types; moreover, there are also differences between the ESXi/vSphere and the Linux/Docker test configurations.

The NFV system was able to automatically recover from faults in all but two cases. We found that when injecting network and storage corruptions in the ESXi/vSphere scenario, the recovery process does not complete due to residual effects of the faults on fault management components, with a significant loss of availability. In these cases, the NFV system would need the manual intervention of a human administrator to restore the failed replicas. These problems are further investigated in the infrastructure-level evaluation (§ V-C).

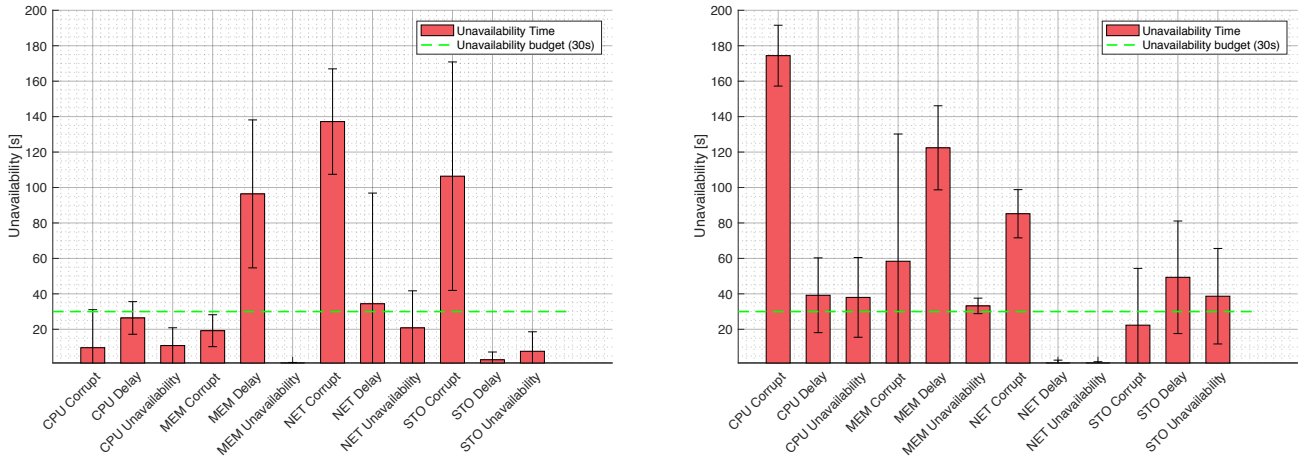
The other fault types show that, in the worst case, the system experiences about 100*s* of unavailability, while in the best cases the VNF unavailability is within the reference value of 30*s*. In particular, in the ESXi/vSphere scenario, storage delay faults have little impact on the system. This behavior is explained by the fact that the virtual disks of VMs are located on a shared iSCSI storage partition, thus, the I/O load on the local physical disk is small, and delay faults are well masked.

CPU delay and unavailability faults show different effects between the ESXi/vSphere and Linux/Docker test configurations. Despite that Linux containers are expected to restart faster (as containers are a lightweight alternative to VMs), the unavailability is higher than ESXi/vSphere. In this case, the recovery time is dominated by a long time-out period that Docker waits for before declaring a node failed. The gap is even worse for CPU corruption faults. In VMware ESXi, any internal kernel error is handled by forcing a crash of the hypervisor, in order to trigger a migration of the workload. Instead, in Linux/Docker, corruption faults did not crash the Linux host, but instead the host continued to execute as much as possible, even in the presence of tasks stalled in kernel space and other internal errors (according to log messages recorded by the kernel). However, the IMS requests on the faulty host experienced errors, and this behavior hinders the migration of containers on the healthy host.

We found another case of incorrect error handling when injecting memory faults. The Linux/Docker scenario shows higher unavailability than ESXi/vSphere, especially in the case of memory delay faults, which overload the memory management subsystem with memory allocations. In this case, the NFVI should protect the VNFs by assuring them enough memory for execution despite the interference of the "hog". Instead, we found that the hog can degrade the performance of the VNFs. Once the injection ends, the NFVI slowly recovers its original performance. The ESXi hypervisor is more robust to these faults since it makes memory reservations for VMs, which cannot be preempted by the hog. Instead, Linux/Docker has a weaker isolation among containers, since it relies on simpler memory management mechanisms, and exposes them to delays caused by excessive swapping and thrashing. Furthermore, during memory unavailability faults, the Linux kernel triggers the *out-of-memory killer* (OOM), which forces the restart of VNFs.

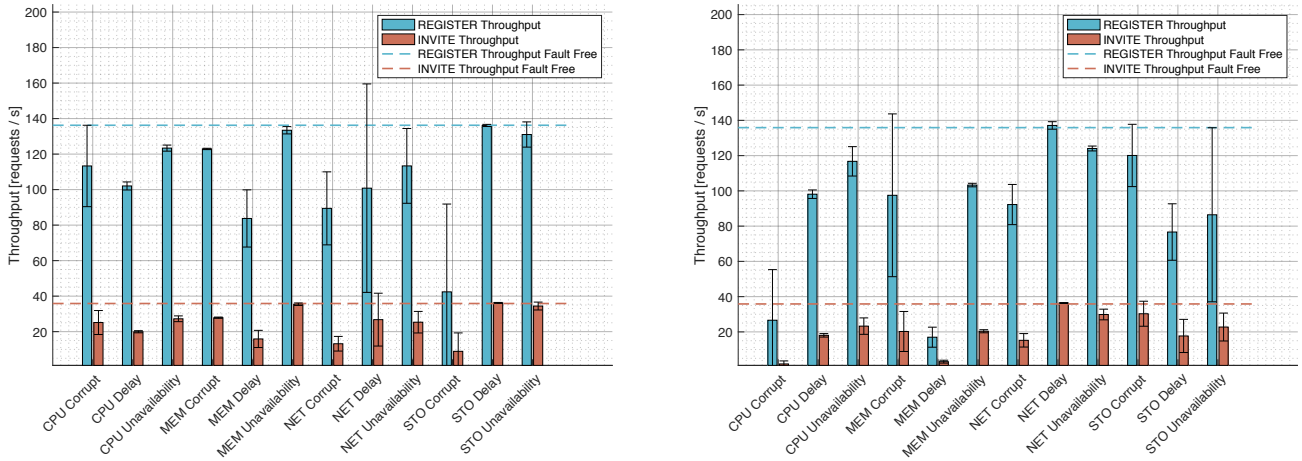
The two NFV configurations also differ with respect to network and storage faults, due to their different architectures: ESXi/vSphere is a more complex configuration, with a higher volume of management traffic over the network (e.g., managing datastores liveness and locking, handling heartbeats, and so on), while Linux/Docker is a more lightweight technology. Furthermore, ESXi and Linux provide different implementations (and thus different vulnerability surfaces) for the TCP/IP protocol, and their log messages denote different reactions to network faults (e.g., packet parsing failures, SYN flooding warnings). Similar considerations also apply for the storage stack, where corruption faults hinder the ESXi hypervisor from creating new threads and performing I/O on the VMFS filesystem. Instead, in Linux/Docker the VNF unavailability is smaller: even if faults cause the crash of some application processes, the system reacts by remounting the filesystem in read-only mode, thus preventing an escalation of the faults. However, Linux/Docker experiences a higher unavailability in the case of storage delays/unavailability, since it generates more I/O traffic on the local disks.

The performance measures, in terms of latency and throughput (Fig. 5 and 6), follow a similar pattern to VNF Unavailability, as the most severe faults (such as network and storage corruptions, and memory delay faults) cause the largest degradation of performance. Moreover, there are differences between ESXi/vSphere and Linux/Docker in terms of perfor-



(a) VMware ESXi/vSphere. (b) Linux/Docker.

Fig. 4: VNF Unavailability under fault injection in the physical layer.



(a) VMware ESXi/vSphere. (b) Linux/Docker.

Fig. 5: VNF throughput under fault injection in the physical layer.

mance. Overall, ESXi/vSphere exhibits better performance in the case of CPU and memory faults. This behavior is due to the ability of hypervisor-based virtualization to guarantee stronger isolation among virtual machines and services. However, Docker provides lower latencies and higher throughput against network faults, since its network stack is more robust and able to quickly recover a good quality of service after fault injection. Regarding storage faults, the performance of ESXi/vSphere is more robust to storage delays/unavailability, but it is exposed to degradation in the case of corruptions.

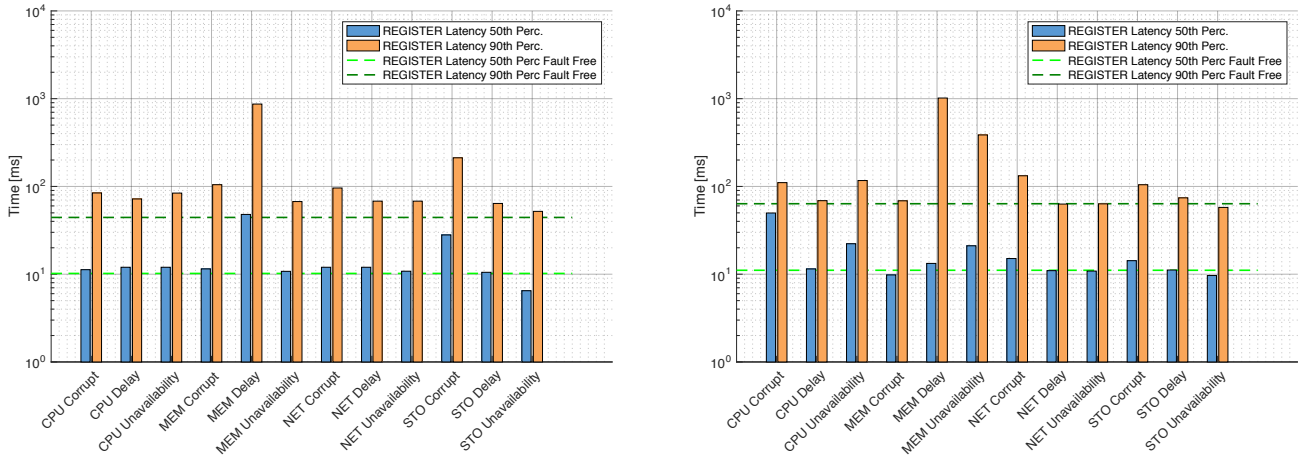
B. Service-level evaluation, faults in the virtual layer

In this section, we evaluate performance and unavailability with respect to faults in the virtual layer of NFVI. Fig. 7 shows the unavailability computed from fault injection experiments respectively on the *Homestead* and *Sprout* VNFs, for both the ESXi/vSphere and Linux/Docker testbeds. For the sake of brevity, we omit the full plots of VNF throughput and latency and include the most relevant observations in the discussion.

Virtual CPU unavailability faults have a higher impact in the case of ESXi/vSphere. Linux/Docker is very quick at recovering from virtual CPU unavailability (i.e., the crash of the VM/container that runs the VNF). As soon as the crash is injected, both Docker and VMware HA recover the VNF by detecting its termination (e.g., an exit with a non-zero status in Linux), and automatically restart it. Restarting a container takes less time than a VM, since it entails to restart a process rather than recreating all the hardware abstractions emulated by the hypervisor and rebooting a full guest OS.

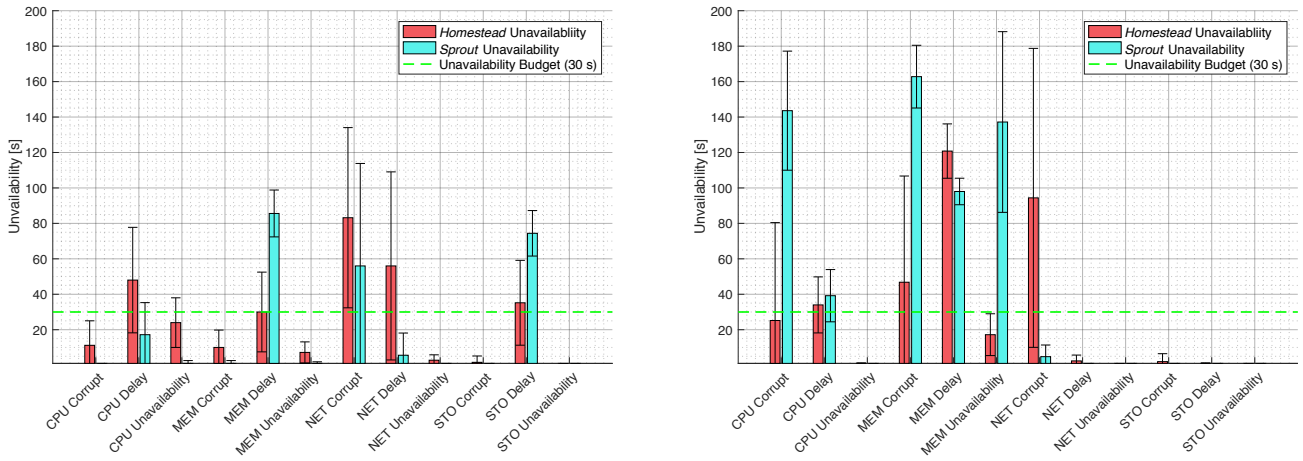
However, different considerations apply for virtual CPU and memory corruption. In the Linux/Docker scenario, these corruptions interfere with the execution of application processes (as the fault corrupts the internal state of VNF software), but the OS does not force the processes to terminate; thus, no recovery is attempted. Instead, in the ESXi/vSphere scenario, a failure is detected as soon as the corruption is perceived by the ESXi hypervisor, and the VM is restarted properly.

As in the case of faults in the physical layer, the Linux/Docker test configuration is more vulnerable to memory



(a) VMware ESXi/vSphere. (b) Linux/Docker.

Fig. 6: VNF Latency under fault injection in the physical layer.



(a) VMware ESXi/vSphere. (b) Linux/Docker.

Fig. 7: VNF Unavailability under fault injection in the *Homestead* and *Sprout* VNFs.

faults. In ESXi/vSphere, when a VMs overloads the memory management subsystem, the hypervisor is able to provide a better isolation among VMs than Linux Docker, thanks to memory reservations and a more efficient memory allocator. Instead, in Linux/Docker, a buggy/overloaded VNF can propagate the performance degradation to other VNFs, as also confirmed by the analysis of throughput and latency of the NFV system as a whole. However, it is worth to point out that ESXi/vSphere is still vulnerable to CPU and memory contention caused by a buggy/overloaded VNF, as the VNF unavailability exceeds in some cases the reference value.

Fault injection in the virtual network shows that the *Homestead* VNF is critical to deliver IMS services. Faults in this VNF cause a noticeable period of unavailability, both in ESXi/vSphere and in Linux/Docker. By analyzing the logs from the infrastructure, we found that the service unavailability was due to disruption of TCP communication (e.g., in the calibration of the TCP window). Network delay faults impact differently on ESXi/vSphere and Linux/Docker due to the different TCP/IP implementations, as ESXi exhibits a higher NIC loss rate due to packets parse failures. Network unavailability faults had a small impact on unavailability (about 3s) compared to network corruption faults since, once the injector does not drop packets anymore, the service performance is restored almost immediately. These considerations also apply for experiments on the *Sprout* node, except that Docker is less affected by network corruptions.

As for corruption and unavailability faults in the virtual storage, the results do not show differences between the ESXi/vSphere and Linux/Docker scenarios in terms of unavailability (Fig. 7), but Linux/Docker has higher VNF latencies (not showed for brevity). This result is the opposite of the case of physical corruption faults: this time, the Linux guest OS inside the ESXi VMs tolerated the fault by remounting the virtual disk as read-only; instead, in the case of Linux/Docker, the partition was managed by Docker, which was not able to remount it. Storage delay faults have a greater impact on ESXi: in this scenario, the VNF generates more load on the virtual disk, as it stores the full VM image with the root filesystem; instead, in Linux/Docker, the virtual disk only

stores the partition with the *Cassandra* datastore. Finally, storage unavailability faults do not have an impact neither in ESXi/vSphere nor in Linux/Docker. The logs from these experiments show that the *Cassandra* instance on the injected virtual node detects an I/O error, and stops itself to prevent further service degradation, leaving the *Cassandra* replica on the other host to serve the requests.

C. Infrastructure-level evaluation

We analyze in detail fault management in the two configurations, by evaluating coverage and latency of fault detection and recovery. These measures are complementary to service-level measures: after a fault, while the network traffic is forwarded to the healthy replicas of the VNFs, the detection and recovery are performed in background to restore the capacity of the NFV system. We use the logs collected from VMware HA and the VMkernel in the ESXi/vSphere scenario, and from the Linux kernel and Docker Swarm in the other one. A fault is detected when there is at least an occurrence of log message related to fault management and to internal errors (e.g., unusual high-severity messages, and messages with specific keywords); and it is considered recovered when there is any specific message that denotes the completion of a recovery action (e.g., restart and migration) and reports that a VM or container is in a running state. The detailed results are summarized in Tables II and III. We focus on fault injection in the physical layer because of space limitations.

TABLE II: Fault detection and fault recovery coverage.

Fault Type	# DETECTED		# RECOVERED				Tot. Exps
	ESXi	Docker	ESXi		Docker		
			MANO	NFVI	MANO	NFVI	
CPU CORRUPT	5	5	5	0	2	0	5
CPU DELAY	5	0	0	5	0	0	5
CPU UNAVAILABILITY	5	5	5	0	5	0	5
MEM CORRUPT	5	5	5	0	5	0	5
MEM DELAY	5	0	0	5	0	0	5
MEM UNAVAILABILITY	5	5	1	4	0	5	5
NET CORRUPT	3	5	1	0	5	0	5
NET DELAY	5	5	1	4	0	5	5
NET UNAVAILABILITY	5	5	0	5	0	5	5
STO CORRUPT	4	5	2	0	3	1	5
STO DELAY	5	0	0	5	0	0	5
STO UNAVAILABILTY	5	5	0	5	5	0	5
Total	57	45	20	33	25	16	60
Percentage	95.00%	75.00%	92.98%		91.11%		

In the ESXi/vSphere scenario, the fault detection coverage is about 95%, within a detection latency of 38.7*s* on average. The fault recovery coverage (successful execution of recovery actions) is also high, except for the cases of storage and network corruptions. In some of the experiments, the fault was tolerated or recovered locally by an NFVI node, with no interaction with the HA manager (these cases are counted in the "NFVI" column of TABLE II, and labeled as *local recovery* in TABLE III). This behavior was observed in the case of CPU and memory delay faults, in which ESXi/vSphere detects an anomalous state during VM operations, by logging that it received intermittent heartbeats from VMs; after fault injection, the nodes are able to locally recover a correct service. The experiments with storage delays and unavailability showed

TABLE III: Fault detection and fault recovery latency.

Fault Type	DETECTION LATENCY [s]		RECOVERY LATENCY [s]	
	ESXi	Docker	ESXi	Docker
CPU CORRUPT	11.3	14.8	66.7	213.8 *
CPU DELAY	58.0	*no detection*	*local recovery*	*not detected*
CPU UNAVAILABILITY	36.8	39.7	48.0	126.8
MEM CORRUPT	45.7	14.8	60.0	104.8
MEM DELAY	49.2	*no detection*	*local recovery*	*not detected*
MEM UNAVAILABILITY	34.7	17.4	136.3	*local recovery*
NET CORRUPT	32.3	18.6	156.7 *	30.2
NET DELAY	92.5	*no detection*	144.5	*local recovery*
NET UNAVAILABILITY	35.2	17.8	*local recovery*	*local recovery*
STO CORRUPT	36.8	15.3	296.2 *	99.7 *
STO DELAY	27.6	*no detection*	*local recovery*	*not detected*
STO UNAVAILABILTY	4.2	16.2	*local recovery*	102.8
Average	38.7	19.3	129.8	91.2

* Latency has been computed only for the recoved cases

a similar behavior. When injecting storage delays, the *Storage I/O Control* (SIOC) module in ESXi reports errors during the usage of the datastore; for storage unavailability faults, ESXi detects that the host datastore is inaccessible. In both cases, the system autonomously recovers after the injection.

Instead, the other experiments required a recovery action from the HA manager, but the recovery could not succeed in some cases. In particular, during network corruption and unavailability fault injection experiments, the injected host is unable to communicate with the other one, and VMware HA detects a *partitioned* state. Then, VMware HA tries to migrate the VMs to the healthy node, but it is forced to cancel the migration (as denoted by log messages such as "*CancelVmPlacement*"), due to residual data corruptions in the persistent state of VMs. In a similar way, in storage corruption faults, the migration of VMs failed to due the corruption of VM data and metadata, which could not be started after the power-off. To avoid these problems, the services and protocols for fault management should be made more robust against corrupted data (e.g., by recognizing invalid data; and by using replicated data to retry migration). Moreover, since these mechanisms are provided by a third-party OTS product, it is important for the designers of the NFV system to discover this kind of vulnerability through fault injection.

It is important to remark that the latency of the recovery process is quite large for ESXi/vSphere, taking on average 129.8*s*. Part of this long time can be attributed to the policy of VMware HA that several *heartbeats* should be unanswered before declaring a node as failed (in the VMware HA terminology, the node goes from *green* to *yellow* state, and then to *red* [72]). Then, VMware HA takes a long time to restart the VMs due to the need for accessing to the shared storage, and to allocate, initialize, and power-on the VM. Unfortunately, this process is too slow for carrier-grade NFV.

In the Linux/Docker scenario, we observe a fault detection coverage of 75%, with a detection latency of 19.3*s* on average. Compared to ESXi/vSphere, there is an improvement with respect to the fault detection latency, but a worse result in terms of fault detection coverage. The reason is that *Docker Swarm* uses a simpler fault detection mechanism, which monitors the network reachability with the hosts, while VMware vSphere combines both network and storage heartbeats and collects diagnostic information from the hosts. Thus, in Linux/Docker,

most of the burden of fault detection is on the host on Linux kernel, which unfortunately provides little information about anomalous states (e.g., as in the case of memory overloads and other delay faults, see § V-A).

Linux/Docker was able to recover most of the faults that were detected, as the fault recovery coverage is 91.11% with a latency of 91.2*s* on average. There are cases in which the fault has been detected but not recovered, such as the CPU corruption experiments: in this case, the injected host is in an anomalous state, but it is not crashed, thus Docker Swarm does not trigger the restart of the containers. In storage corruption fault injection, Docker Swarm did not migrate the containers because the fault did not impact on the host network communication, thus the host was considered alive even if the fault impacted on service availability (see also Fig. 4b).

VI. Discussion and conclusion

Delivering a reliable NFV system is a challenging problem, as it entails several design decisions for configuring fault management policies and selecting third-party virtualization and management products. Often, it is not clear how a choice will impact on performance and availability, and how to get a good trade-off. Thus, we proposed a dependability benchmark for the direct measurement of dependability and performance of an NFV system, and presented a case study on two major virtualization paradigms.

Our experiments point out several useful findings. Despite the promise of higher performance and manageability, container-based virtualization can be less dependable than the hypervisors. The ESXi/vSphere configuration showed a higher fault detection coverage, due to more sophisticated fault management mechanisms than Linux/Docker Swarm, which is a relatively less mature technology. The NFV system designers should compensate for these limitations, by pairing Docker with additional solutions for detecting problems not reported by the OS, such as memory overloads, and by configuring recovery actions for specific symptoms, such as internal kernel errors and I/O errors. For example, cloud monitoring dashboards, such as Datadog [74] and Librato [75], allow to setup and customize policies to detect fault symptoms (such as resource utilization peaks and trends) and to trigger maintenance tasks (e.g., scaling or rebooting); the Linux kernel can be configured (e.g., at compile time) to adopt a more conservative behavior in the case of internal errors, by forcing a reboot in order to trigger the fault management process; I/O errors can be prevented by adopting redundant or more reliable I/O interfaces.

Another advantage of the benchmark is that it allows NFV designers to tune the configuration, and to repeat experiments to measure and validate improvements in terms of coverage and latency of fault detection and recovery. For example, our experiments pointed out that the speed of fault management in ESXi/vSphere could be improved by tuning the *heartbeat* period and the time-to-reboot of VMs.

In this work, we have focused on comparing two different NFV setups using alternative virtualization paradigms (container- and hypervisor-based), but the dependability benchmark is also applicable for other types of comparisons, such as: to compare different VNFs (e.g., alternative IMS products) using the same NFVI and virtualization technology; to consider different virtualization technologies that adopt the same virtualization paradigm (such as VMware ESXi versus Xen or KVM), or different physical setups (e.g., by varying the number and type of hardware machines); and to compare different MANO products. The purpose of this work has been to provide a general and flexible methodology suitable for benchmarking different NFV designs.

References

[1] ETSI, "Network Functions Virtualization - An Introduction, Benefits, Enablers, Challenges & Call for Action," Tech. Rep., 2012.
[2] ——, "Network Functions Virtualisation (NFV) - Network Operator Perspectives on Industry Progress," Tech. Rep., 2013.
[3] A. Manzalini, R. Minerva, E. Kaempfer, F. Callegari *et al.*, "Manifesto of edge ICT fabric," in *Proc. ICIN*, 2013.
[4] Technavio, *Global Network Function Virtualization Market 2016-2020.* , 2016.
[5] SDNCentral LLC., *2016 Mega NFV Report*. https://www.sdxcentral.com/reports/nfv-vnf-2016-download/, 2016.
[6] ETSI, "Network Function Virtualisation (NFV) - Use Cases," Tech. Rep., 2013.
[7] Quality Excellence for Suppliers of Telecommunications Forum (QuEST Forum), "TL 9000 Quality Management System Measurements Handbook 4.5," Tech. Rep., 2010.
[8] ETSI, "GS NFV-REL 001 - V1.1.1 - Network Functions Virtualisation (NFV); Resiliency Requirements," 2015.
[9] H. S. Gunawi, A. Laksono, R. O. Suminto, M. Hao *et al.*, "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages," in *Proc. SoCC*, 2016.
[10] TPC Council. Homepage. http://www.tpc.org/.
[11] SPEC. Homepage. https://www.spec.org/.
[12] DBench Project, "Project website," http://www.laas.fr/DBench/, 2004.
[13] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society, 2008.
[14] M. Vieira and H. Madeira, "Benchmarking the Dependability of Different OLTP Systems," in *Proc. DSN*, 2003.
[15] J.-C. Ruiz, P. Yuste, P. Gil, and L. Lemus, "On benchmarking the dependability of automotive engine control applications," in *Proc. DSN*, 2004.
[16] ENISA, "Cloud computing certification." [Online]. Available: https://resilience.enisa.europa.eu/cloud-computing-certification
[17] Clearwater, "Project Clearwater - IMS in the Cloud," 2014. [Online]. Available: http://www.projectclearwater.org/
[18] C. Rotter, L. Farkas, G. Nyíri, G. Csatári, L. Jánosi, and R. Springer, "Using Linux Containers in Telecom Applications," *Proc. ICIN*, 2016.
[19] R. Cziva, S. Jouet, K. J. White, and D. P. Pezaros, "Container-based network function virtualization for software-defined networks," in *Proc. ISCC*, 2015.
[20] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, 2007.
[21] A. Mukherjee and D. Siewiorek, "Measuring software dependability by robustness benchmarking," *IEEE TSE*, vol. 23, no. 6, 1997.
[22] ISO/IEC, "ISO/IEC 25010:2011, Systems and software Quality Requirements and Evaluation (SQuaRE)," 2011.
[23] R. Jain, *The art of computer systems performance analysis*. John Wiley & Sons, 1990.
[24] J. Gray, "Why Do Computers Stop and What Can Be Done About It?" in *Proc. SRDS*, 1985.
[25] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems," in *Proc. FTCS*, 1991.
[26] J. Durães and H. Madeira, "Generic Faultloads based on Software Faults for Dependability Benchmarking," in *Proc. DSN*, 2004.
[27] A. B. Brown, L. C. Chung, and D. A. Patterson, "Including the human factor in dependability benchmarks," in *Proc. DSN*, 2002.
[28] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" in *USENIX Symp. on Internet Technologies and Systems*, 2003.
[29] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, 1997.
[30] S. Winter, T. Piper, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "GRINDER: on reusability of fault injection tools," in *Proc. AST*, 2015.
[31] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, 2016.

[32] K. Wolter, A. Avritzer, M. Vieira, and A. Van Moorsel, *Resilience assessment and evaluation of computing systems*. Springer, 2012.
[33] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE TSE*, vol. 16, no. 2, 1990.
[34] K. Kanoun, M. Kaâniche, and J.-P. Laprie, "Qualitative and quantitative reliability assessment," *IEEE Software*, vol. 14, no. 2, pp. 77–87, 1997.
[35] W. H. Sanders and J. F. Meyer, "Stochastic activity networks: Formal definitions and concepts," in *Lectures on Formal Methods and PerformanceAnalysis*. Springer Berlin Heidelberg, 2001, pp. 315–343.
[36] P. Koopman and J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems," *IEEE TSE*, vol. 26, no. 9, 2000.
[37] J. Durães, M. Vieira, and H. Madeira, "Multidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior," *IEICE Trans. on IS*, vol. 86, no. 12, 2003.
[38] A. Albinet, J. Arlat, and J. Fabre, "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel," in *Proc. DSN*, 2004.
[39] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A Framework for Cloud Recovery Testing," in *Proc. NSDI*, 2011.
[40] P. Joshi, H. S. Gunawi, and K. Sen, "Prefail: A programmable tool for multiple-failure injection," in *Proc. OOPSLA*, 2011.
[41] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva, "On fault resilience of OpenStack," in *Proc. SoCC*, 2013.
[42] Netflix, "The Chaos Monkey." [Online]. Available: https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey
[43] C. Pham, D. Chen, Z. Kalbarczyk, and R. K. Iyer, "CloudVal: A framework for validation of virtualization environment in cloud infrastructure," in *Proc. DSN*, 2011.
[44] F. Cerveira, R. Barbosa, H. Madeira, and F. Araujo, "Recovery for Virtualized Environments," in *Proc. EDCC*, 2015, pp. 25–36.
[45] D. Cotroneo, L. De Simone, A. K. Iannillo, A. Lanzaro, and R. Natella, "Dependability evaluation and benchmarking of network function virtualization infrastructures," in *Proc. NetSoft*, 2015.
[46] A. Sunyaev and S. Schneider, "Cloud services certification," *Communications of the ACM*, vol. 56, no. 2, 2013.
[47] Cloud Watch HUB, "Cloud certification guidelines and recommendations." [Online]. Available: www.cloudwatchhub.eu
[48] L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approach*, 5th ed. Morgan Kaufmann Publishers Inc., 2011.
[49] E. Bauer and R. Adams, *Reliability and Availability of Cloud Computing*, 1st ed. Wiley-IEEE Press, 2012.
[50] ETSI, "Network Function Virtualisation Infrastructure Architecture - Overview," Tech. Rep., 2014.
[51] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, 1991.
[52] D. Powell, "Failure mode assumptions and assumption coverage," in *FTCS*, vol. 92, 1992.
[53] M. Le and Y. Tamir, "Fault injection in virtualized systems—challenges and applications," *IEEE TDSC*, vol. 12, no. 3, 2015.
[54] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo, "Virtual CPU validation," in *Proc. SOSP*, 2015.
[55] Amazon.com, Inc. (2011, Apr.) Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. [Online]. Available: http://aws.amazon.com/message/65648/
[56] A. Warren. (2011, Sep.) What Happened to Google Docs on Wednesday. [Online]. Available: http://googleenterprise.blogspot.it/2011/09/what-happened-wednesday.html
[57] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, "Fault Injection Experiments using FIAT," *IEEE Trans. Comp.*, vol. 39, no. 4, 1990.
[58] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A tool for the validation of system dependability properties," in *Proc. FTCS*.
[59] T. Tsai and R. Iyer, "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," in *Proc. MMB*, 1995.
[60] S. Han, K. Shin, and H. Rosenberg, "DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment," in *Proc. CPDS*, 1995.
[61] J. Arlat, J. Fabre, M. Rodríguez, and F. Salles, "Dependability of COTS Microkernel-Based Systems," *IEEE TC*, 2002.
[62] L. De Simone, "Dependability Benchmarking of Network Function Virtualization," Ph.D. dissertation, Univ. of Naples Federico II, 2017.
[63] ETSI, "Network Functions Virtualisation (NFV); Assurance; Report on Active Monitoring and Failure Detection ," Tech. Rep., 2016.
[64] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. SoCC*, 2010.
[65] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the Weather Tomorrow?: Towards a Benchmark for the Cloud," in *Proc. DBTest*, 2009.
[66] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *Proc. CCA*, 2008.
[67] ETSI, "Report on the application of Different Virtualization Technologies," Tech. Rep., 2016.
[68] VMware Inc., "Delivering High Availability in Carrier Grade NFV Infrastructures," *White Paper. VMware vCloud NFV*, 2010.
[69] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *Proc. ICNC*, 2016.
[70] A. Lakshman and P. Malik, "Cassandra," *SIGOPS Operating Systems Review*, vol. 44, no. 2, Apr. 2010.
[71] Gayraud, R. and Jacques, O. and Day, R. and Wright, C. P. SIPp. http://sipp.sourceforge.net/.
[72] VMware Inc. (2016) vSphere Virtual Machine Administration. https://www.vmware.com/support/pubs/.
[73] Docker Inc. Docker Swarm. https://www.docker.com/products/docker-swarm.
[74] Datadog Inc. DatadogHQ Homepage. https://www.datadoghq.com/.
[75] Librato Inc. Librato Homepage. https://www.librato.com/.

Domenico Cotroneo (Ph.D.) is associate professor at the Federico II University of Naples. His main interests include software fault injection, dependability assessment, and field measurement techniques. He has been member of the steering committee and general chair of the IEEE Intl. Symp. on Software Reliability Engineering (ISSRE), PC co-chair of the 46th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN), and PC member for several scientific conferences on dependable computing including SRDS, EDCC, PRDC, LADC, SafeComp.

Luigi De Simone received his MSc degree with honors in Computer Engineering in 2013, and the PhD degree from the Federico II University of Naples, Italy, working on reliability evaluation of Network Function Virtualization infrastructures, within the Dependable Systems and Software Engineering Research Team (DESSERT) group. His research activity focuses on fault injection and dependability benchmarking of operating systems and cloud computing infrastructures. He received the "Best Student Presentation Award" from the ISSRE 2014 Conference, and the "Best Paper Award" from the NetSoft 2015 Conference.

Roberto Natella (Ph.D.) is a postdoctoral researcher at the Federico II University of Naples, Italy, and co-founder of the Critiware s.r.l. spin-off company. His research interests include dependability benchmarking, software fault injection, and software aging and rejuvenation, and their application in operating systems and virtualization technologies. He has been involved in projects with Leonardo-Finmeccanica, CRITICAL Software, and Huawei Technologies. He contributed, as author and reviewer, to several leading journals and conferences on dependable computing and software engineering, and he has been organizing the workshop on software certification (*WoSoCer*) within the IEEE ISSRE conference.