Enhancing Failure Propagation Analysis in Cloud Computing Systems

Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella
Università degli Studi di Napoli Federico II
Naples, Italy
{cotroneo, luigi.desimone, pietro.liguori, roberto.natella}@unina.it

Nematollah Bidokhti
Futurewei Technologies, Inc.
USA
nbidokht@futurewei.com

***Abstract*—In order to plan for failure recovery, the designers of cloud systems need to understand how their system can potentially fail. Unfortunately, analyzing the failure behavior of such systems can be very difficult and time-consuming, due to the large volume of events, non-determinism, and reuse of third-party components. To address these issues, we propose a novel approach that joins fault injection with anomaly detection to identify the symptoms of failures. We evaluated the proposed approach in the context of the OpenStack cloud computing platform. We show that our model can significantly improve the accuracy of failure analysis in terms of false positives and negatives, with a low computational cost.**

***Index Terms*—Fault Injection; Failure Analysis; Anomaly Detection; Cloud Computing Systems; OpenStack; Debugging**

I. Introduction

Cloud computing is becoming an attractive solution for running services with high-reliability requirements, such as in the telecom and healthcare domains [1]–[4]. However, cloud computing systems are often exposed to unpredictable failure conditions [5]. These failures can propagate across several components or layers of the system (e.g., storage, virtual network, compute instances, etc.) in complex ways, leading to cascading effects (*failure propagation*) that make recovery actions more problematic.

Therefore, identifying and analyzing failure propagation is an important activity to design more effective recovery actions. Fault injection is a relevant approach, which emulates faults to anticipate worst-case scenarios, such as network partitions, high network latency, replica crashes, and I/O exceptions [6]–[11]. Fault injection has reached a level of maturity that it is routinely used to reveal failures in real-world systems, including cloud computing software such as key-value data stores and distributed computing frameworks (e.g., Cassandra, ZooKeeper) [9], entire cloud computing services (e.g., streaming services deployed by Netflix) [12] and infrastructures (e.g., IaaS providers such as Amazon) [13].

Nevertheless, there are still open issues for its adoption in cloud systems. Indeed, as the scale and the complexity of these systems increase, it becomes harder for developers to identify (and to analyze) failures that are triggered by fault injection. Furthermore, failure propagation analysis too often relies on the knowledge, the experience, and the intuition of human analysts since existing fault injection solutions provide limited support to the analyst for understanding what happened during an experiment [14].

The current state of practice is to detect failures (e.g., service unavailability, performance degradation) by monitoring the quality of service during the fault injection test; more sophisticated solutions detect failures by monitoring properties expressed with formal specifications, such as finite state machines [15], relational logic [9], and special-purpose languages [16]. However, once a service failure has been triggered by fault injection and detected by monitoring mechanisms, a human analyst still needs to analyze the chain of events (e.g., messages) that occurred among the location where the fault/error is injected and the component that experiences the service failure. Yet, this failure analysis still relies on intuition and manual effort of the human analyst [17]. Unfortunately, manual analysis is too difficult and time-consuming, because of:

- The **high volume of messages** generated by large distributed systems that the human analyst needs to scrutinize;
- The **non-determinism in distributed systems**, in which the timing and the order of messages can unpredictably change even if there is no failure, which introduces noise in the analysis, and increases the effort of the human analyst to pinpoint the failure (i.e., to discriminate the anomalies caused by a fault from genuine variations of the system);
- The use of **"off-the-shelf" software components**, either proprietary or open-source (such as application frameworks, middleware, data stores, etc.), whose events and protocols can be difficult to understand and to manually analyze.

This work aims to provide automated support for analyzing failures triggered by fault injection in cloud computing systems. We aim to avoid the human analyst to manually inspect thousands of events, by automatically identifying the few relevant events that are related to the injected fault, while discarding noisy, uninteresting events. To this goal, we propose an approach that extends fault injection, by combining it with black-box tracing and anomaly detection for failure analysis. The driving idea is to train a *probabilistic model* of the events in the distributed system under test under *fault-free* conditions, by using variable-order Markov Models for analyzing event sequences. Afterward, the system is tested with fault injection, and event traces are collected under these *faulty* conditions. The faulty event traces are analyzed with anomaly detection by using the probabilistic model, and the anomalous events are reported to the human analyst for understanding how to avoid failures.

We experimentally evaluate the proposed approach in the context of the OpenStack cloud management platform, which is the basis for many commercial cloud management products [18], and it is widespread both among public cloud infrastructure providers and private users [19]. Our experiments show that the proposed approach can be applied to event traces that are generated by a large, "off-the-shelf" distributed system, without relying on knowledge of its internals, with a low rate of false

978-1-7281-4982-0/19/$31.00 ©2019 IEEE
DOI 10.1109/ISSRE.2019.00023

IEEE computer society

positives (i.e., genuine variations are not mistaken for failure symptoms) and of false negatives (i.e., actual anomalies caused by a fault are not missed), and with a low computational cost.

In the following of this paper, Section II elaborates on the problem addressed by this paper, and provides a motivating example; Section III presents the proposed methodology for failure analysis; Section IV experimentally evaluates the methodology; Section V discusses related work; Section VI concludes the paper.

II. Problem Statement

To better understand the research problem addressed by this paper, we discuss an example of fault-injection experiment on the OpenStack cloud computing platform. OpenStack is a cloud management system that controls large pools of computing, storage, and networking resources in a data center. It provides a dashboard and APIs that can be used both by cloud operators to manage the infrastructure, and by end-users to offer resources as-a-service.

In our fault-injection experiments, we inject faults into the three most important services of OpenStack [20], [21]: (i) the **Nova** subsystem, which provides services for provisioning instances (VMs) and handling their life cycle; (ii) the **Cinder** subsystem, which provides services for managing block storage for virtual instances; and (iii) the **Neutron** subsystem, which provides services for provisioning virtual networks, including resources such as *floating IPs*, *ports* and *subnets* for instances. In turn, these subsystem includes several components (e.g., the Nova sub-system includes *nova-api*, *nova-compute*, etc.), which interact through message queues internally to OpenStack. The Nova, Cinder, and Neutron sub-systems provide external REST API interfaces to cloud users.

A simple graphical representation of a fault-injection experiment is shown in Fig. 1. This representation shows remote procedure calls that are made for communication in the distributed system. These calls are displayed as intervals over the timeline of the experiment. We consider both API calls between the client and the OpenStack REST APIs (the topmost sequence of calls), and internal API calls within OpenStack, which are performed by Nova, Neutron, and Cinder using message queues (the other three sequences of calls). In order to see the effects of the injected fault, we show two subplots: the former shows a normal execution of the system (*fault-free execution*), in which no fault is injected; the latter shows the execution of the system when a fault is injected in the Nova subsystem (*faulty execution*). Since both executions are performed under the same conditions (i.e., same software and hardware configuration, same workload, etc.), any deviation between the faulty and the fault-free execution is considered an anomaly due to the injected fault.

The workload used in this example first creates several resources (i..e, networks, instances, volumes, etc.), then it performs basic operations in order to stimulate the different components of the system (e.g., attaching a volume to an instance, check the connectivity, reboot an instance, etc.) before cleaning up the created resources. All these operations are performed by invoking the OpenStack APIs.

One of these API calls is an asynchronous request for creating a new VM instance. After the API call ends, OpenStack Nova takes a few minutes for creating and initializing the instance. During these operations, we inject a Python exception in order to force a failure (Ⓐ).

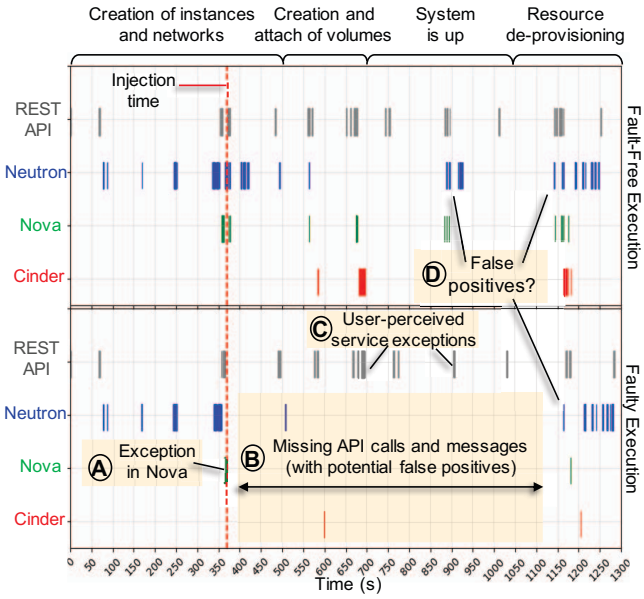


Fig. 1. A graphical representation of a fault-injection experiment.

Fig. 1 points out that there are several API calls in the fault-free execution that are missing in the faulty execution (Ⓑ) since the injected fault causes a failure that affects several OpenStack subsystems over a relatively long time period. Indeed, Nova does not complete the initialization of the VM instance due to the fault, leaving the VM in an inactive state. Moreover, the OpenStack Neutron subsystem was also unable to attach the virtual network to the VM instance. Later on (i.e., after about five minutes) the workload client experienced a service exception when calling the API of the Cinder subsystem, which manages storage volumes in OpenStack (Ⓒ). Consequently, the workload could not attach the volume to the VM instance. Both Nova and Neutron do not raise any API exception, but the failure only became apparent to the client when invoking the API of the Cinder subsystem. Therefore, the issue propagates both across subsystems (from Nova to Neutron and Cinder) and across time, since the client perceives the failure only after a relatively long time. This behavior is problematic from the point of view of high-availability, and thus of defining proper recovery actions, as the propagation delay also increases the time-to-detect and the time-to-recover the failure. Furthermore, the longer the propagation chain the more difficult will be for a developer reasoning about how to best tolerate the fault, e.g., whether to manage the fault in Nova, Neutron and/or Cinder and at which time to manage the fault during the workload. For example, the API could return a more timely notification of the failure to the client, either by introducing a callback mechanism in the Nova API that creates the instance or by returning an error from other API calls to Nova or Neutron.

The analysis of a fault-injection experiment can be inaccurate due to the non-determinism of the API calls in distributed systems. For example, the Neutron subsystem uses asynchronous messages and polling for distributing state updates across its components, thus such messages could be easily misclassified as anomalies. Moreover, due to the asynchronous nature of several APIs, it is difficult to properly identify whether API calls order does not matter (i.e., is due to non-determinism) or should be carefully taken into account because of the failure. In this

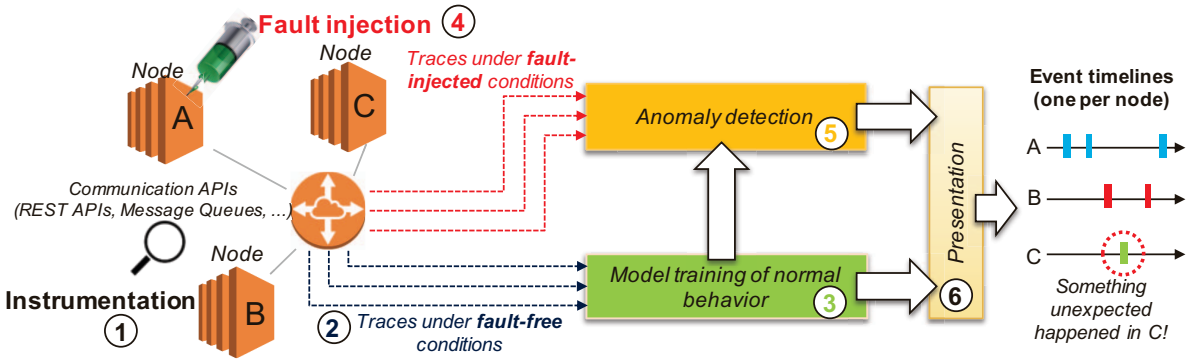


Fig. 2. Overview of the proposed approach.

point, Fig. 1 also highlights events that could be false positives (Ⓓ), both among the fault-free and the faulty execution. Thus, we need to understand if the differences among such two executions are due to the non-determinism in the system (i.e., they are not related to the failure) or not (i.e., they are actually anomalies). Considering the false positives makes the debugging more difficult and cumbersome for the human analyst, as each execution may include hundreds of API calls to analyze with only a few ones relevant for understanding the failure.

In this work, we propose an algorithm for enhancing the failure propagation analysis of the fault-injection experiments, by adopting a rigorous probabilistic approach to pinpoint unlikely messages that are related to the failure, with the goal of achieving high accuracy in identifying the true anomalies.

III. Proposed Methodology

Fig. 2 shows an overview of the approach. Firstly, we instrument the communication APIs of the system (step ①). We consider a distributed system as a set of black-box components that interact with each other via public service interfaces (e.g., REST APIs, message queues). Therefore, we exercise the system by applying a workload without injecting any fault (step ②). We record all messages exchanged among the components, and between the components and the workload client. These messages constitute the *fault-free trace*. Several fault-free traces are collected by executing the same workload several times, to take into account the natural variability of such traces.

In order to have an accurate model of fault-free system behavior, we define a probabilistic model that is trained by the fault-free traces (step ③). Due to non-determinism, this model considers the "benign" variability of the interactions (e.g., different ordering, type, or duration of events) that can occur under fault-free conditions. After training the model, a fault injection experiment is performed in the distributed system (step ④), for each fault encompassed in the analysis. This step will produce the so-called *fault-injected traces* (also *faulty traces*), i.e., one per experiment. The faulty traces are then analyzed by the proposed probabilistic model in order to detect the actual deviation(s), i.e., the anomaly (ies), from the normal behavior (step ⑤).

In order to emphasize messages that were omitted because of the injected fault (i.e. only occurring in fault-free conditions), and new messages that were caused by the injected fault (i.e., only occurring under faulty conditions), the results of anomaly detection are visualized by presenting to the human analyst the messages of both the fault-injected and of a fault-free execution (step ⑥).

Figure 3 shows a detailed flowchart of the proposed approach. In the following of this section, we discuss the individual steps of the workflow.

A. Instrumentation and pre-processing

The proposed approach records and analyzes *messages* that occur in the distributed system. Messages are the key observation point for debugging and verification of distributed systems, as they reflect well the activity of the distributed system [22]. For example, nodes perform work when they receive message requests (e.g., through remote procedure calls), and reply with messages for providing responses and results; moreover, nodes use messages to asynchronously notify a new state to other nodes in the distributed system.

Thus, the first step of our approach consists in instrumenting the distributed system under test, in order to keep track of the messages that are sent between nodes during a test. In general, it is possible to get traces of the communication among components leveraging *run-time tracing techniques*, which allow us to instrument the source- or binary-code and record the execution of specific points in the software. In particular, our approach gets information about messages by collecting traces of *communication API invocations* made by the distributed software. For example, in our approach we instrument the calls to APIs of popular middleware technologies such as *REST frameworks* (e.g., Django [23] and Spring [24]) and *message queueing* (e.g., AMQP [25] and RabbitMQ [26]). Example of tracing toolkits are Zipkin [27] (used in this paper), Jaeger [28], and Appdash [29]. These monitoring tools are familiar to developers of distributed systems, as they are already used for debugging, performance monitoring and optimization, root cause analysis, and service dependency analysis [30], [31].

This instrumentation is a form of "black-box tracing" since it does not require any knowledge about the internals of the system under test, but only which are the communication APIs used by the system. This approach is suitable when testers do not have a full and detailed understanding of the entire distributed system; this is the case of distributed systems developed by large teams (in which testers and developers might be distinct people), and distributed systems that embed components developed by third-parties.

The approach records the beginning and the end of every call to the communication APIs by inserting a probe using the distributed tracing system. The tracer records all information about the exchanged messages, such as the time at which the communication

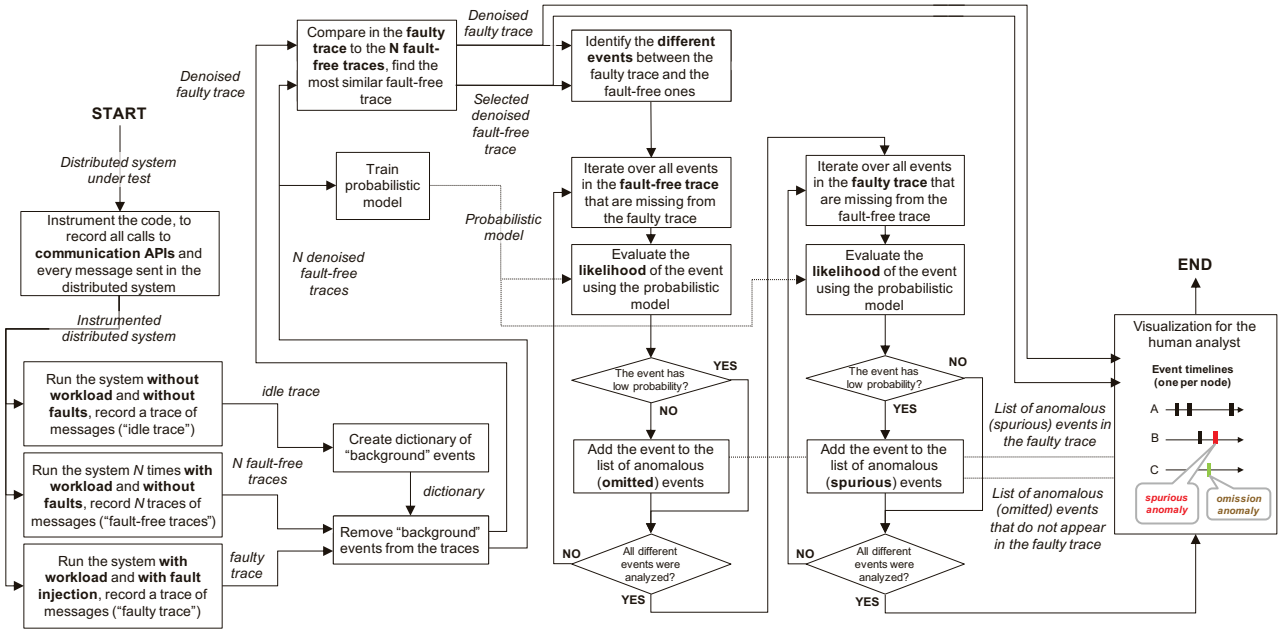


Fig. 3. Detailed workflow of the proposed approach.

API has been called and its duration, the component that invoked the API (*message sender*), and the remote service that has been requested through the API call (*called service*). We refer to the calls to communication APIs as *events*; thus, the execution of the distributed system generates an *event trace*. The approach orders the events in the trace with respect to the timestamp of the event collector. Our anomaly detection technique is designed to be tolerant to the non-determinism (e.g., due to random messaging delays) of the events by using a probabilistic technique, which will be discussed in the section III-D.

This lightweight approach for event collection allows us to deploy tracing with low intrusiveness and does not require more detailed information about the internals of the system. For example, the tracer within the components does not need to collect and propagate a session identifier across messages related to the same session, which would require the human analyst to customize the data collection according to the specific application [32] or to collect more extensive information at the OS- and network-level [33], [34].

B. Data collection

Once the distributed system has been instrumented, it is executed several times to perform fault injection tests. The distributed system is monitored during test execution; at the same time, the system is stimulated with a workload (e.g., by generating client requests), and a fault is injected into the system. Each test injects a different fault, and only one fault is injected per test. For each test, we collect a message trace (**fault-injected trace**).

In addition to fault-injected traces, we also execute the system and collect traces without fault injection (**fault-free traces**). In general, collecting fault-free traces (also known as *golden runs* or *reference runs*) is a common practice in fault injection experiments since they are used as a reference to understand how the system derailed from a correct execution due to the injected fault [14], [35], [36]. While previous studies used this approach for the analysis of non-distributed systems (such as embedded systems), we extend this approach to distributed systems, by addressing the problems of non-determinism and scalability of the analysis. To collect fault-free traces, the approach executes the system N times, by running the same workload used in fault injection tests, but without injecting any fault. The messages exchanged in each execution are stored in a fault-free trace, i.e., one fault-free trace per workload execution. These N fault-free traces are then used for training the model of "normal" behavior of the distributed system. The model will be used as a reference for analyzing failures. We use more than one fault-free trace since the model needs to reflect the variability of the execution that characterizes distributed systems (e.g., the relative ordering of messages). We expect that the larger is the number of training traces, the more accurate is the model that represents the normal behavior. The use of the fault-free traces is discussed more in detail in the next subsections, and the impact of the number of training traces is empirically evaluated in Section IV.

Finally, during data collection, we need to take into account the messages that are not due to the workload but they are independently generated by background and asynchronous activities in the system at arbitrary times. For example, these messages represent events that are internally produced by garbage collection, resource monitoring, updating database indexes, etc.. Since these messages are not strictly related to the workload, they can (mistakenly) appear as anomalies during fault injection tests. For this reason, our approach properly removes such unrelated events. Therefore, we perform a preliminary analysis of the system in which no workload is applied. The approach keeps the system idle for a few minutes before and after a fault-free execution of the workload and records any background message into a trace (**idle trace**). Then, we use the idle trace to create a dictionary of *background events* that will be ignored in the subsequent analysis of the fault-free and fault-injected traces.

C. Trace comparison

Internally, the approach represents the events within a trace with unique identifiers (i.e., *symbols*), so that two events of the same type are identified by the same symbol. In particular, we assign a unique symbol to every distinct pair <*message sender*, *called service*> (e.g., <*Cinder*, *attach volume*>). Thus, the event traces are converted into *sequences of symbols*.

In order to identify the differences between the faulty and the normal execution of the system, the approach performs a string comparison on the fault-injected sequence and one of the fault-free sequences. In particular, the approach looks for the *longest common subsequence* (LCS) of such sequences [37]. The LCS is a subset of symbols that are present in both sequences in the same order, and that can be obtained by removing (a minimal number of) symbols from the original sequences. This kind of problem is recurrent in computer science, such as in bioinformatics and in source code versioning (e.g., in the *diff* Unix tool), and can be solved with efficient algorithms [38], [39].

To perform the comparison, the approach selects one fault-free trace among the ones collected at the beginning of the workflow (Fig. 3). In particular, the approach selects the fault-free trace *most similar* to the fault-injected trace since we want to identify and to filter out from the failure analysis as much *common* events as possible (i.e., the approach aims to discard the subset of messages that also happen with the same type and order in at least one fault-free sequence), in order to focus the attention of the human analyst on the *anomalous* events (i.e., the differences between the faulty and the most similar fault-free sequence).

The similarity between two strings x and y is measured by considering the length of the LCS ($|LCS(x,y)|$) [40], i.e., the number of symbols that appear in both strings while preserving the order of symbols. In particular, we compute the normalized length of the LCS ($nLCS$), where $nLCS(x,y) = \frac{|LCS(x,y)|}{\sqrt{l_x \cdot l_y}}$ and where l_x and l_y are the lengths of the individual strings x and y. The approach uses this metric to identify the fault-free trace of the training set most similar to the fault-injected trace (*selected fault-free trace*).

D. Probabilistic modeling

The analysis performed with LCS is still prone to inaccuracies since there may be differences between the fault-injected trace and the selected fault-free trace that are caused by non-deterministic reorderings, and thus are not related to failures. These differences lead to *false positives* that may divert the attention of the human analyst. To overcome this problem, the approach uses a *Markov model* to estimate the probability of an event, in order to evaluate whether the event is anomalous in a probabilistic sense. Markov modeling is a popular approach for the probabilistic analysis of sequences of symbols (e.g., to predict the probability of a future symbol), such as in bioinformatics [41], data compression [42], and text and speech recognition [43]. In our context, we evaluate the probability of the events marked as anomalies in the previous comparison performed with LCS. Thus, we use the probabilistic model as a further reference to analyze the anomalous events. Such a model takes into account the "benign" variations in the ordering and type of messages that happen in fault-free conditions.

We opt for a Markov model where the states are a direct representation of the observed events. However, a simple Markov chain still does not suffice for our purposes, since the probability of the next state (i.e., the next event of the sequence) would only depend on the current state (i.e., the *memoryless* property). In general, this is not the case for a sequence of events that can be generated by a distributed system; in practice, the probability of an event is correlated with the history of the previous events. For example, in the case of the OpenStack platform, the occurrence of an event representing a "volume attach" operation must be preceded by a sequence of several preliminary operations on the volume and on the instance to be attached (e.g., an instance must be created and initialized before attaching volume).

Ultimately, we decide to use *higher-order* Markov models, where the probability of events takes into account the history of the previous states of a sequence. In particular, since conditioning random variables could vary based on the specific observed realization, we adopt *Variable-order Markov Models* (VMMs). VMMs estimate the probability that a symbol σ can appear after a sequence s (named *context*), by counting the joint occurrences of σ and s in the training sequence to build the predictor $\hat{P}$, for variable cardinalities of s [44].

In this work, we use the notation defined by Begleiter et al. [44]. Let Σ be a finite alphabet. A learner is given a training sequence $x_1^n = x_1 x_2 ... x_n$, where $x_i \in \Sigma$ and $x_i x_{i+1}$ is the concatenation of x_i and x_{i+1}. Based on x_1^n, the goal is to learn a model $\hat{P}$ that provides a probability assignment for any future outcome given the past. Specifically, for any context $s \in \Sigma$ and symbol $\sigma \in \Sigma$, the learner should generate a conditional probability $\hat{P}(\sigma|s)$. The accuracy of the predictor $\hat{P}(\cdot|\cdot)$ is typically measured by its average log-loss $l(\hat{P}, x_i^T)$ with respect to a test sequence $x_1^T = x_1 ... x_T$:

$$\ell(\hat{P}, x_i^T) = -\frac{1}{T}\sum_{i=1}^{T} \log \hat{P}(x_i | x_1 ... x_{i-1}) \quad (1)$$

There exist many algorithms in the scientific literature for training and applying VMMs [44]. Our approach uses the *Prediction by Partial Matching - Method C* (PPM-C) lossless compression algorithm [45], which is a variant of the original PPM algorithm published in 1984 by Cleary and Witten [46] that includes a set of improvements proposed by Moffat [47]. PPM is a finite-context statistical modeling technique that builds a predictor by combining several fixed-order context models [45], with different values of the order, ranging from zero to an upper bound D (i.e., the maximal order of the Markov model) [48]. For more detailed information on PPM and the Method C variant, we refer the reader to the work of Begleiter et al. [44].

In this work, we set the maximum order of the VMMs, i.e. D, by measuring the number of events that can be triggered by an individual request from a client of the distributed system. In response to a client request, the distributed system generates a sequence of messages among its internal components, until it reaches again a quiescent state, or it returns a reply to the client. Since an event is most likely influenced by the previous events in the context of the same client request, we set D to the maximum number of events triggered by a client request. This choice is conservative since this number (e.g., several tens of events in our case study) tends to be much higher than the context length chosen in previous studies on VMMs [45].

E. Classification procedure

The ultimate result of the proposed approach is to classify the events into:

- **Common events**: Events that occurred both in the fault-injected trace and in *at least one* of the fault-free traces, with the same type and order.
- **Anomalous events**: Differences between the fault-injected trace and all of the fault-free traces. These events are further classified into:
  - **Spurious events**: Events that would normally not happen under fault-free conditions.
  - **Missing events**: Events that happen in fault-free conditions, but do not happen under fault injection.

The approach trains the VMM by using a set of $n-1$ fault-free traces (i.e., all the fault-free traces, except the *selected fault-free trace* with the highest similarity to the fault-injected trace). Then, we apply the VMM to compute the probabilities of events, in order to determine whether they are anomalous. Specifically, the approach performs two steps:

▷ *Analysis of LCS differences that only appear in the fault-injected trace.* In the first step, the fault-injected trace takes the role of the *test sequence* for the VMM. We focus on symbols of the test sequence that were highlighted as differences in the previous LCS analysis. The goal is to confirm whether these symbols are actually unlikely events, not only with respect to the selected fault-free trace (i.e., the one used for determining the LCS) but also according to the whole set of fault-free traces in the training set. For each event not included in the LCS, we compute the probability of the event according to the VMM. If the probability is lower than a threshold $\epsilon_{\text{SPURIOUS}}$, then the symbol has a low likelihood to appear in that position of the sequence; thus, the VMM confirms that the symbol represents a **spurious** anomalous event. Otherwise, the event is considered non-anomalous.

▷ *Analysis of LCS differences that only appear in the selected fault-free trace.* In the second step, the *selected fault-free trace* takes the role of the *test sequence* for the VMM. As for the previous step, we focus on symbols of the test sequence that were highlighted as differences in the previous LCS analysis. In this case, we consider the events that only appear in the selected fault-free trace: therefore, from the point of view of the fault-injected trace, these events represent *omissions*. This step confirms whether these omissions are indeed likely, and thus should be considered anomalies. The approach applies the VMM to the events that only appear in the fault-free trace, by computing the probabilities of such events according to remaining fault-free traces in the dataset. If the probability of the event is higher than a threshold $\epsilon_{\text{MISSING}}$, then there is a high likelihood for the symbol to be in that position of the sequence. Therefore, the fact that the event is missing in the fault-injected trace should be considered an anomaly, and thus it is marked as a **missing** anomalous event. Otherwise, if the probability of the event is not high, then the lack of the event from the fault-injected trace is considered non-anomalous.

We remark that even if the two steps perform similar comparisons, the results obtained by them are different and complementary. If the fault-injected trace contains an anomalous event with a *low probability value* according to the VMM, then it is confirmed as spurious. Similarly, if the fault-injected trace does not contain an event with a *high probability value* in the selected fault-free trace, then the event is confirmed to be an omission. A practical approach is to select conservative thresholds (e.g., $\epsilon_{\text{SPURIOUS}} = 20\%$ and $\epsilon_{\text{MISSING}} = 80\%$), so that the VMM can filter out most of the LCS differences that are not actually spurious/missing events; and to leave to the human analyst the decision about the uncertain events. Therefore, the accuracy of the probabilistic model is an important factor that makes this approach suitable in practice. The accuracy of our approach is further analyzed in the rest of the paper.

IV. Experimental Evaluation

In this section, we evaluate the approach in the context of fault injection experiments in the OpenStack platform. In § IV-A, we present the experimental setup, and in § IV-B and § IV-C we report on the accuracy and performance of the proposed approach.

A. Experimental Setup

In our fault-injection experiments, we targeted OpenStack version 3.12.1 (release *Pike*), deployed on Intel Xeon servers (E5-2630L v3 @ 1.80GHz) with 16 GB RAM, 150 GB of disk storage, and Linux CentOS v7.0, connected through a Gigabit Ethernet LAN.

We injected faults during the execution of OpenStack components, by simulating exceptional conditions during the interactions between components. We targeted the internal APIs used by OpenStack components for managing instances, volumes, networks, and other resources. For example, we injected faults during calls to the *nova-compute* component within the Nova subsystem to manage new instances. The injected faults represent exceptional cases, e.g., a resource that is not found or unavailable, a processing delay when retrieving a resource, or an incorrect value caused by the user, the configuration, or a bug inside OpenStack. In particular, we considered the following kind of faults:

- **Throw exception**: An exception is raised on a method call, according to pre-defined, per-API list of exceptions;
- **Wrong return value**: A method returns an incorrect value. In particular, the returned value is corrupted according to its data type (e.g., we replace an object reference with a null reference, or replace an integer value with a negative one);
- **Wrong parameter value**: A method is called with an incorrect input parameter. Input parameters are corrupted according to the data type, as for the previous fault type;
- **Delay**: A method is blocked for a long time before returning a result to the caller. This fault can trigger timeout mechanisms inside OpenStack or can cause a stall.

We performed three distinct fault injection campaigns, in which we applied three different workloads described in the following.

▷ **New deployment workload** (DEPL): This workload configures a new virtual infrastructure from scratch, by stimulating all of the target subsystems (i.e., Nova, Neutron, and Cinder) in a balanced way. This workload creates VM instances, along with key pairs and a security group; attaches the instances to an existing volume; creates a virtual network consisting in a subnet and a virtual router; assigns a floating IP to connect the instances to the virtual network; reboots the instances, and then deletes them;

▷ **Network management workload** (NET): This workload includes network management operations, in order to stress

more the Neutron subsystem and virtual networking. The workload initially creates a network and a VM, then generates network traffic via the public network. After that, it creates a new network with no gateway, brings up a new network interface within the instance, and generates traffic to check whether the interface is reachable. Finally, it performs a router rescheduling, by removing and adding a virtual router resource;

▷ **Storage management workload** (STO): This workload performs storage management operations on instances and volumes, in order to stress more the Nova and Cinder subsystems. In particular, the workload creates a new volume from an image, boots an instance, then rebuilds the instance with a new image (e.g., as it would happen for an update of the image). Finally, it performs a cleanup of the resources.

All of these workloads invoke the OpenStack APIs, which are provided by the Nova, Cinder, and Neutron subsystems. We implemented the workloads by reusing integration test cases from the *OpenStack Tempest* project [49], since these tests are already designed to trigger several subsystems and components of OpenStack and their virtual resources. We selected this kind of workload in order to point out propagation effects across subsystems that may be caused by fault injection.

In-between calls to service APIs, our workload generator performs *assertion checks* on the status of the virtual resources, in order to reveal failures of the cloud management system. In particular, these checks assess the connectivity of the instances through SSH and query the OpenStack API to ensure that the status of the instances, volumes, and the network is consistent with the expectation of the tests. In the context of our methodology, assertion checks serve as *ground truth* about the occurrence of failures during the experiments (i.e., a reference for evaluating the accuracy of the proposed approach). We consider an experiment as failed if at least one API call returns an error (**API error**) or if there is at least one assertion check failure (**assertion check failure**). Before every experiment, we clean-up any potential residual effect from the previous experiment, in order to ensure that the potential failure is only due to the current injected fault. To this end, we re-deploy the cloud management system, remove all temporary files and processes, and restore the OpenStack database to its initial state.

In order to find all the injectable locations in Nova, Neutron, and Cinder, we performed a full scan of the source code according to the fault types described above. Then, for each workload, we identified the injectable locations that were covered by the workloads (i.e., we run the workload without injecting anything), and we performed one fault injection test per covered location. In total, we performed $2{,}137$ fault injection tests, and we observed failures in $1{,}432$ tests (67%). In the remaining tests (33%), there were neither API errors nor assertion failures, since the fault did not affect the behavior of the system (e.g., the corrupted state is not used in the rest of the experiment). This is a typical phenomenon that often occurs in fault injection experiments [50], [51]; yet, the experiments provided us a large and diverse set of failures for our analysis.

Table I shows, for each workload, the number of *unique events* (i.e., the events with different pair <*message sender*, *called service*>) observed in the distributed system during the execution of the workloads, the average length of the fault-free sequences (in term of number of events in the trace), the total number of fault injection experiments for the workload, and the number of experiments that experienced at least one failure. The number of unique events and the total number of events reflects the extent and diversity of the work put on the system. We notice that DEPL is the most extensive workload in terms of both distinct operations and the total number of operations, followed by NET and by STO. These differences among the workloads are meant to evaluate the approach under different levels of complexity and non-determinism.

TABLE I
WORKLOAD CHARACTERISTICS

Workload	Num. unique events	Avg. num. of events per fault-free trace	Num. of total exps.	Num. of failed exps.
DEPL	53	243	945	449
NET	37	212	290	206
STO	34	85	902	777

We used the distributed tracer *Zipkin* for collecting message traces. We instrumented the following communication points:

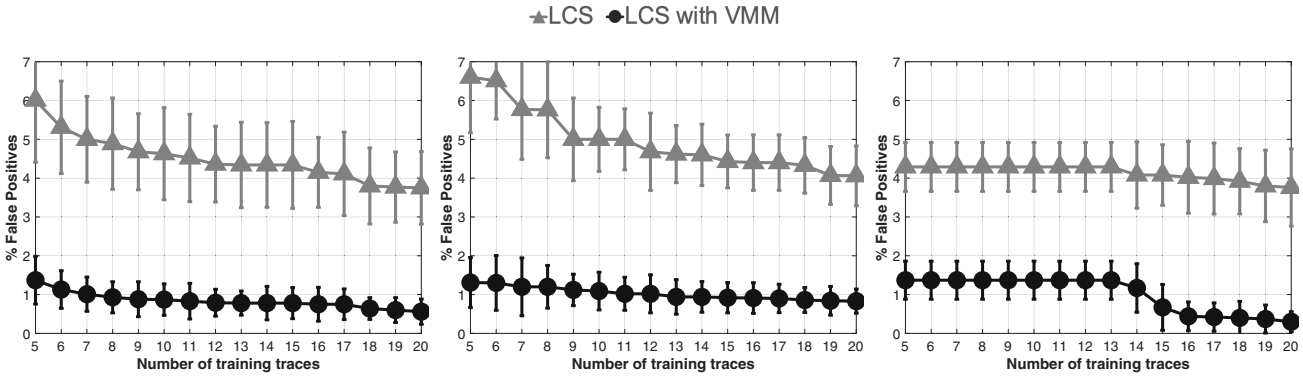
- The *OSLO Messaging library*, which uses a message queue library, by exchanging messages with an intermediary queuing server (RabbitMQ) through RPC messages. These messages are used for communication among OpenStack subsystems;
- The *RESTful API libraries* of each OpenStack subsystem, i.e., the *novaclient* for Nova (it implements the OpenStack Compute API [52]), the *neutronclient* for Neutron (it implements the OpenStack Network API [53]), and the *cinderclient* for Cinder (it implements the OpenStack Block Storage API [54]). These interfaces are used for communication between OpenStack and its clients.

In total, we instrumented only 5 selected functions of these components (e.g., the `cast` method of OSLO to broadcast messages), by adding very simple annotations only at the beginning of these methods, for a total of 20 lines of code. We neither added any further instrumentation to the subsystems under test nor used any knowledge about OpenStack internals.

B. Accuracy evaluation

We evaluated the accuracy of the proposed approach in terms of *false positives* and *false negatives*. False positives are non-anomalous events that are mistakenly labeled as anomalous (either spurious or missing) by the proposed approach. False negatives are anomalous events that are not identified by the approach (i.e., they are labeled as non-anomalous).

Our experiments generated about half a million events across more than two thousands of execution traces. A key concern for evaluating anomaly detection is the need for a reliable ground truth about the actual label of the events (anomalous or non-anomalous). Unfortunately, manually assigning labels to such a large set of data is prone to errors and unfeasible in practice. Thus, we adopt an automated approach. In order to understand which suspicious event (i.e., spurious or missing events) is not actually anomalous, we performed an analysis using an increasing number of sequences of distinct fault-free executions. Since such executions represent the normal behavior



(a) New deployment workload (DEPL). (b) Network management workload (NET). (c) Storage management workload (STO).

Fig. 4. False positives analysis.

TABLE II
FALSE NEGATIVES ANALYSIS.

Workload type	LCS	LCS with VMM
DEPL	6.35%	7.00%
NET	0%	0.91%
STO	0.88%	1.37%

of the system, every anomaly identified by the approach should be considered as a false positive.

As a term of comparison, we consider both the full approach, denoted as "LCS with VMM", and a baseline approach denoted as "LCS". The "LCS" approach represents a simplistic approach to failure analysis that just aligns and compares traces without using a probabilistic model to account for non-deterministic variations. In this way, we can separately evaluate the relative impact on the accuracy of LCS and of the VMM.

For each workload, we collect a set of fault-free traces an order of magnitude larger than the set of trace used to train the model. Then, we randomly choose n traces to train the model before evaluating the approach with other m distinct fault-free traces. The n training traces and the m test traces are always disjoint sets. We vary the number of training traces (i.e., $n=5,6,...,20$) in order to evaluate the impact of the size of the training set on the accuracy of the approach. Furthermore, we perform tens of repetition for each fixed value of n, with different random selections of the training traces and of the test traces. For each repetition, we compute the percentage of anomalies (either spurious or missing events) with respect to the length of the compared sequences. This provides us a metric to evaluate the ratio of the false alarms over the total number of events.

Figure 4 shows, for each workload, how the average percentage of false positives varies with the number n of training traces. For each data point, the sub-figures show a vertical error bar representing the standard deviation of the percentage of false positives across repeated evaluations. We found that increasing the number n of training traces brings an incremental reduction of the percentage of false positives. In all cases, the percentage settles around 1% on average for "LCS with VMM". The simpler "LCS" approach has a higher percentage of false positives, which exceeds the 6% in the case of the first two workloads, and 4% in the case of the third workload.

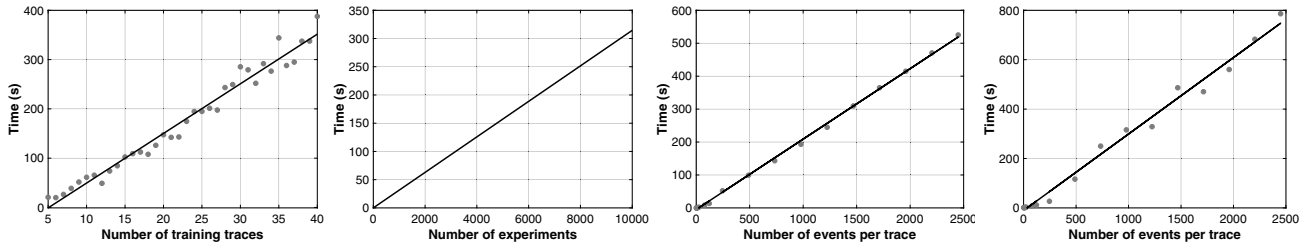
We can also see that the "LCS with VMM" approach is less sensitive to the size of the set of training traces when the workload is very extensive (i.e., DEPL and NET). Indeed, the average percentage of false positives is reduced by at most 1% with respect to the case of a small training set ($n=5$), while the "LCS" exhibits a wider variation. Our approach provides a percentage of false positives almost stable for all the number of training traces, regardless of the workload; moreover, the uncertainty intervals are overlapping for all values of n. The lower sensitivity to the size of the training set makes the "LCS with VMM" approach more predictable and easier to apply in practice.

Overall, the curves point out that the VMM can improve accuracy, especially for lower sizes of the training set. The difference of accuracy between the VMM and the plain LCS is wider when the number of events in the workload is higher (DEPL and NET): in fact, in these cases, the LCS string comparison technique generates a high percentage of false positives due to the difficulty at aligning a large number of events with several differences in the middle of the traces. In the case of a workload with a shorter number of events (STO), the LCS analysis provides a lower percentage of false positives and, thus, the gap with the VMM is less pronounced.

To evaluate false negatives, we focus on the experiments that experienced at least a failure. We remark that we consider an experiment as failed if at least one API returns an error or if there is at least an assertion check failure during the execution of the workload. Since the VMM is applied in pipeline after the LCS, there is a risk that the VMM misclassifies an anomalous event as non-anomalous, thus neglecting the failure-related events (i.e., a false negative). In the ideal case, the percentage of false negatives for the VMM matches the LCS. We expect that, if the VMM is accurate enough, the percentages of false negatives for the VMM and for the LCS should be very close.

Before applying the LCS and VMM approaches, we conservatively remove all the uncertain event types that were marked as false positives in the previous analysis in at least one case. Afterward, if at least one of the remaining events is identified as an anomaly, then we consider such an event as a true anomaly. Otherwise, if there is no true anomaly, we consider the experiment as a case of a false negative. We evaluate the false negatives by measuring the percentage of failures with no anomalies reported over all the experiment that experienced a failure.

This method of evaluation is very conservative since we are entirely removing event types that could lead to false positives. Even if these events could have represented true anomalies in



(a) Training time while increasing the number of the training traces.
(b) Classification time while increasing the number of the experiments.
(c) Training time while increasing the size of the traces.
(d) Classification time while increasing the size of the traces.

Fig. 5. Computational cost of the proposed approach.

some experiments, we ignore anomalies raised for these events, thus restricting the chances of the VMM to point out the failure. However, this approach assures that we do not over-estimate the ability of the VMM at identifying true anomalies since we only take into account anomalies for events that were never affected by false alarms in our previous extensive analysis.

We executed this analysis for each workload and for different choices of the number n of training traces ($n = 5,10,15,20$). Table II shows the percentage of experiments that experienced a failure, and where the failure was not detected by the approach. This metric is computed by evaluating the number of failed tests in which the algorithm points out no true anomalies for the experiment. The metric is not necessarily 1 for the "LCS" approach since there were failures in our fault injection experiments in which there were neither omitted nor spurious messages (e.g., 6.35% of experiments in DEPL). This behavior happened in the case of "local" failures of individual OpenStack components, which did not perform the expected job, but still sent and received the same set of messages of fault-free runs.

We found that the percentage does not vary for different values of n (therefore, the table only reports one value per configuration). Moreover, we found that "LCS" and "VMM" provide similar results (the differences is always lower than 1% for all the workloads). It is important to recall that this is an ideal case: since VMM is applied in cascade after LCS, the VMM cannot identify new failures beyond the "suspect" events pointed out by LCS. Instead, there is a risk that the VMM filters out some of these events, potentially causing false negatives. However, this is never the case, as the "VMM" always raises at least one true positive, even if few anomalies are mistaken as false negatives. This result highlights that the proposed approach can avoid many false positives with a negligible risk of missing a test with a failure. This result is valuable for human analysts since they can focus their debugging activities on a few, specific events that are actually failure-related.

Moreover, we looked in detail at the log messages that were generated by the tests for which we had false negatives. We found that false negatives only occur in failures with no propagation across components. For example, a failure during the cleanup of a resource at the end of the workload typically does not affect any other subsequent operation. For this reason, there is a higher percentage of false negatives in the DEPL workload, since this workload creates (and, thus, deletes) more resources. Conversely, the approach reveals 0.91% cases of false negatives in the NET workload since it does not contain any cleanup operation, suggesting that the approach is very accurate in terms of false negatives.

C. Performance evaluation

We evaluated the computational cost of the proposed approach, by measuring the time taken for analyzing the event traces, both at training and at classifying them. We performed the analysis with respect to increasing volumes of data, i.e., by varying the number of traces to analyze and the number of the events per-trace (i.e., the length of the sequences of symbols).

Figure 5a shows how the time to train the model grows with the number of training traces. As described in the Subsection IV-B, when the number of training traces increases, the accuracy of the training model improves. However, a large number of fault-free traces increases both the time for executing more fault-free runs, as well as the computational time for the data analysis, with an approximately linear trend. However, the computational cost imposed by VMM seems small enough for practical purposes, since the duration of the data analysis is close to 5 minutes for up to 40 training traces.

Figure 5b shows the time for processing an increasing number of fault injection tests (up to 10,000 fault injection tests), for a fixed number and size of training traces ($N = 10$). The duration increases linearly, but with a lower trend than the previous cases. This is due to the fact that most of the computational cost of the VMM algorithm comes from the training phase, while the estimation phase takes a relatively small amount of time. Therefore, the VMM approach can scale well for high numbers of fault injection tests.

Figure 5c gives an indication of the time to train the model when the number of events per trace increases, for a fixed number of training traces ($N = 10$). The computational cost also grows linearly with the length of the traces, with a slope similar to the previous analyses. In the worst case of thousands of events per trace, the duration of the training can grow up to about 6 minutes, which is still a reasonable duration.

Finally, Figure 5d shows the duration of event classification, when the number of events per trace increases, for a fixed number of traces to analyze (2,000 traces). In the worst case, the duration of the training can grow up to about 13 minutes.

V. Related Work

Research studies on debugging distributed systems lead to a variety of *profiling* techniques to pinpoint bugs and performance bottlenecks. Aguilera et al. [55] collect black-box network traces of communications between hosts, in order to analyze requests as they move through the system (e.g., web requests across the tiers of a web application). Their approach infers causal paths of the requests, by tracing call pairs (i.e., request messages,

and their corresponding responses), and by analyzing statistical correlations. However, this approach focuses on synchronous (RPC-style) interactions between components, and it is not meant to analyze asynchronous interactions (i.e., the server immediately replies to a request, before issuing causally-related requests and performing more work) and rare events (as the approach focus on the most frequent interactions).

Magpie [34] and Pinpoint [56] reconstruct causal paths by using more sophisticated tracing infrastructures, by tracing detailed events at the OS-level and at the application server level. The tracing tags incoming requests with a unique *path identifier*, and associates resource usage throughout the system with that identifier. This fine-grain tracing approach does not rely on statistical inference and can provide high accuracy, but it also brings considerable complexity, which makes it difficult to deploy it in practice, especially when considering cloud computing infrastructures with many heterogeneous components (e.g., OSes, middleware, interpreters, etc.).

Gu at al. [33] proposes a methodology to extract knowledge on distributed system behavior of request processing without source code or prior knowledge. The authors construct the distributed system's component architecture in request processing and discover the heartbeat mechanisms of target distributed systems.

Pip [16] is a system for automatically checking the behavior of a distributed system against programmer-written expectations about the system. Pip provides a domain-specific expectations language for writing declarative descriptions of the expected behavior of large distributed systems and relies on user-written annotations of the source code of the system to gather events and to propagate path identifiers across chains of requests. This approach provides flexibility for the analysis but requires access to the source code, and non-negligible efforts to annotate it.

More recent studies contributed to tools resembling debuggers, but for distributed systems. Pensieve [57] is an approach for producing the path to failure, in a similar way to delta debugging: it combines static analysis, and re-execution of the system with iteratively-refined logging, in order reconstruct the intermediate path backward from the failure to the user inputs and events that cause the failure. Friday [58] is a distributed debugger that allows developers to replay a failed execution of a distributed system, and to inspect the execution through breakpoints, watchpoints, single-stepping, etc., at the global-state level. ShizViz [59] is an interactive tool for visualizing execution traces of distributed systems, which allows developers to intuitively explore the traces and to perform searches; moreover, the tool provides support for comparing distributed executions with a pairwise comparison, even if without probabilistic techniques to filter-out benign variations due to non-determinism.

Recent fault injection solution addressed cloud computing systems. The *Fate* [9] tool, and its successor *PreFail* [10], simulate disk failures, network partitions, and crashes of nodes, by exploring multiple occurrences of faults during the same experiment, to test recovery procedures more thoroughly (e.g., at tolerating further network/disk faults occurring during recovery). To address the combinatorial explosion of experiments, these tools adopt user-programmable policies to prune redundant experiments (e.g., injections in symmetric states or in paths that were already covered). Ju et al. [6], *ChaosMonkey* [12], and *Jepsen* [60] 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* [7] and Cerveira et al. [11] use fault injection (CPU and memory corruptions, resource leaks) to test the isolation among hypervisors and VMs. Pham et al. [8] applied fault injection on OpenStack to create signatures of the failures, in order to support problem diagnosis when the same failures happen in production. Once fault injection reveals a failure, in most cases it is the tester's responsibility to look at what happened during the test, and come up with an interpretation of the issue and of a potential solution to make the system more fault-tolerant.

Our approach differs from anomaly detection solutions using ML models or employing self-adapted monitoring [61]–[63], and it is unique in the design space of distributed debugging tools. To the best of our knowledge, this is the first approach that applies distributed debugging techniques for interpreting the fault injection experiments. In the context of fault injection, the fault-free executions are used as a reference for identifying anomalies in fault-injected executions performed under the same conditions (same workload, same node deployment, etc.): therefore, the approach does not rely on programmer-written specifications to identify failures (even if such specifications could cooperate with our approach to gain further insights); moreover, our approach does not rely on inferring causal relationships (which requires more intrusive instrumentation and may be inaccurate for asynchronous and rare interactions). Since the approach only relies on modeling the observed sequences of events, it can be easily deployed and integrated into interactive tools for debugging and visualization, to provide more robust trace comparison and analysis abilities.

VI. Conclusion

In this paper, we propose a technique for analyzing execution traces of distributed systems under fault injection, by comparing the executions to fault-free ones in order to point out anomalies. To address the problem of non-determinism (which may lead to "benign" anomalies not actually related to failures) we develop a sequence comparison approach supported by a probabilistic model. The probabilistic model is built from a group of several fault-free execution traces, in order to reflect "benign" variations that normally occur in the distributed system. Moreover, to make the approach applicable to black-box systems and not reliant on intrusive instrumentation, we base our probabilistic model only on externally-observable traces of messages, which are analyzed as sequences of symbols using Variable-order Markov Models. We evaluated the approach within the OpenStack cloud computing platform: we found that the VMM limits the false positives compared to a non-probabilistic comparison of execution sequences, without significant loss in terms of false negatives. Moreover, the VMM is lightweight enough to be applicable with a low computational cost. Future development is to integrate the approach with tools for fault injection and debugging, such as for reporting the anomalies to the users and for clustering fault injection tests to better support the human analysts.

Acknowledgments

This work has been partially supported by the PRIN 2015 project "GAUSS" funded by MIUR (Grant n. 2015KWREMX_002) and by UniNA and Compagnia di San Paolo in the frame of Programme STAR.

REFERENCES

[1] N. Sultan, "Making use of cloud computing for healthcare provision: Opportunities and challenges," *International Journal of Information Management*, vol. 34, no. 2, pp. 177–184, 2014.

[2] M.-H. Kuo, "Opportunities and challenges of cloud computing to improve health care services," *Journal of medical Internet research*, vol. 13, no. 3, p. e67, 2011.

[3] C. Doukas and I. Maglogiannis, "Bringing iot and cloud computing towards pervasive healthcare," in *Proc. of the 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IEEE, 2012, pp. 922–926.

[4] Z. Yin, F. R. Yu, S. Bu, and Z. Han, "Joint cloud and wireless networks operations in mobile cloud computing environments with telecom operator cloud," *IEEE Transactions on Wireless Communications*, vol. 14, no. 7, pp. 4020–4033, 2015.

[5] P. Garraghan, R. Yang, Z. Wen, A. Romanovsky, J. Xu, R. Buyya, and R. Ranjan, "Emergent failures: Rethinking cloud reliability at scale," *IEEE Cloud Computing*, vol. 5, no. 5, pp. 12–21, Sep. 2018.

[6] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva, "On fault resilience of OpenStack," in *Proc. SoCC*, 2013.

[7] 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.

[8] C. Pham, L. Wang, B.-C. Tak, S. Baset, C. Tang, Z. T. Kalbarczyk, and R. K. Iyer, "Failure diagnosis for distributed systems using targeted fault injection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 503–516, 2017.

[9] 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, pp. 238–252.

[10] P. Joshi, H. S. Gunawi, and K. Sen, "Prefail: A programmable tool for multiple-failure injection," in *Proc. OOPSLA*, 2011.

[11] F. Cerveira, R. Barbosa, H. Madeira, and F. Araujo, "Recovery for Virtualized Environments," in *Proc. EDCC*, 2015, pp. 25–36.

[12] Netflix, "The Chaos Monkey," 2017. [Online]. Available: https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey

[13] T. Limoncelli, J. Robbins, K. Krishnan, and J. Allspaw, "Resilience engineering: learning to embrace failure," *Communications of the ACM*, vol. 55, no. 11, pp. 40–47, 2012.

[14] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM CSUR*, vol. 48, no. 3, p. 44, 2016.

[15] P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal *et al.*, "Uncovering bugs in distributed storage systems during testing (not in production!)." in *Proc. FAST*, 2016, pp. 249–262.

[16] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *Proc. NSDI*, vol. 6, 2006, pp. 9–9.

[17] D. Cotroneo, L. De Simone, A. Di Martino, P. Liguori, and R. Natella, "Enhancing the analysis of error propagation and failure modes in cloud systems." in *ISSRE Workshops*, 2018, pp. 140–141.

[18] OpenStack project, "The OpenStack marketplace," 2018. [Online]. Available: https://www.openstack.org/marketplace/distros/

[19] ——, "User stories showing how the world #RunsOnOpenStack," 2018. [Online]. Available: https://www.openstack.org/user-stories/

[20] J. Denton, *Learning OpenStack Networking*. Packt Publishing Ltd, 2015.

[21] M. Solberg, *OpenStack for Architects*. Packt Publishing, 2017.

[22] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 517–530, 2016.

[23] Django. Home page of Django. [Online]. Available: https://www.djangoproject.com

[24] Spring. Home page of Spring Framework. [Online]. Available: https://spring.io/projects/spring-framework

[25] AMPQ. Home page of AMPQ. [Online]. Available: https://www.amqp.org/

[26] Pivotal. Home page of RabbitMQ. [Online]. Available: https://www.rabbitmq.com/

[27] Zipkin. Home page of Zipkin. [Online]. Available: https://zipkin.io

[28] Jaeger. Home page of Jaeger. [Online]. Available: https://www.jaegertracing.io

[29] Appdash. Home page of Appdash. [Online]. Available: https://github.com/sourcegraph/appdash

[30] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services." in *Proc. OSDI*, 2014, pp. 217–231.

[31] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. DSN*. IEEE, 2002, p. 595.

[32] OpenStack. osprofiler. [Online]. Available: https://docs.openstack.org/osprofiler/latest/user/background.html

[33] J. Gu, L. Wang, Y. Yang, and Y. Li, "Kerep: Experience in extracting knowledge on distributed system behavior through request execution path," in *Proc. ISSREW*. IEEE, 2018, pp. 30–35.

[34] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems." in *Proc. HotOS*, 2003, pp. 85–90.

[35] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[36] M. Leeke and A. Jhumka, "Evaluating the use of reference run models in fault injection analysis," in *Proc. PRDC*. IEEE, 2009, pp. 121–124.

[37] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proc. SPIRE*. IEEE, 2000, pp. 39–48.

[38] J. W. Hunt and M. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[39] E. W. Myers, "An O (ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.

[40] S. Budalakoti, A. N. Srivastava, M. E. Otey *et al.*, "Anomaly detection and diagnosis algorithms for discrete symbol sequences with applications to airline safety," *IEEE Trans. on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 39, no. 1, p. 101, 2009.

[41] M. Stanke and S. Waack, "Gene prediction with a hidden markov model and a new intron submodel," *Bioinformatics*, vol. 19, no. suppl_2, pp. ii215–ii225, 2003.

[42] J. Rissanen, "A universal data compression system," *IEEE Transactions on information theory*, vol. 29, no. 5, pp. 656–664, 1983.

[43] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proc. of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[44] R. Begleiter, R. El-Yaniv, and G. Yona, "On prediction using variable order markov models," *Journal of Artificial Intelligence Research*, vol. 22, pp. 385–421, 2004.

[45] J. G. Cleary and W. J. Teahan, "Unbounded length contexts for ppm," *The Computer Journal*, vol. 40, no. 2_and_3, pp. 67–75, 1997.

[46] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. on Communications*, vol. 32, no. 4, pp. 396–402, 1984.

[47] A. Moffat, "Implementing the ppm data compression scheme," *IEEE Trans. on Communications*, vol. 38, no. 11, pp. 1917–1921, 1990.

[48] S. M. Mavadati, H. Feng, A. Gutierrez, and M. H. Mahoor, "Comparing the gaze responses of children with autism and typically developed individuals in human-robot interaction," in *Proc. HUMANOIDS*. IEEE, 2014, pp. 1128–1133.

[49] OpenStack, "Tempest Testing Project," 2018. [Online]. Available: https://docs.openstack.org/tempest

[50] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*. IEEE, 1996, pp. 304–313.

[51] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An empirical study of injected versus actual interface errors," in *Proc. International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 397–408.

[52] OpenStack. Home page of OpenStack Compute API. [Online]. Available: https://developer.openstack.org/api-ref/compute/

[53] ——. Home page of OpenStack Network API. [Online]. Available: https://developer.openstack.org/api-ref/network/v2/

[54] ——. Home page of OpenStack Block Storage API. [Online]. Available: https://developer.openstack.org/api-ref/block-storage/

[55] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 74–89, 2003.

[56] Y.-Y. M. Chen, A. J. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, "Path-based failure and evolution management," in *Proc. NSDI*, 2004, pp. 309–322.

[57] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan, "Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach," in *Proc. of the SOSP*. ACM, 2017, pp. 19–33.

[58] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global comprehension for distributed replay," in *Proc. NSDI*, vol. 7, 2007, pp. 285–298.

[59] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Queue*, vol. 14, no. 2, p. 50, 2016.

[60] K. Kingsbury, "Jepsen: A framework for distributed systems verification, with fault injection," 2018. [Online]. Available: https://github.com/jepsen-io/jepsen

[61] J. Alonso, L. Belanche, and D. R. Avresky, "Predicting software anomalies using machine learning techniques," in *Proc. of the IEEE 10th International Symposium on Network Computing and Applications*. IEEE, 2011, pp. 163–170.

[62] C. Sauvanaud, K. Lazri, M. Kaâniche, and K. Kanoun, "Anomaly detection and root cause localization in virtual network functions," in *Proc. of the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 196–206.

[63] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring, "Self-adaptive software system monitoring for performance anomaly localization," in *Proc. of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 197–200.