2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)

# Dependability Stress Testing of Cloud Infrastructures

Lena Feinbube*, Lukas Pirl*, Peter Tröger†, Andreas Polze*
* *Operating Systems and Middleware Group, Hasso Plattner Institute at University of Potsdam*
{*firstname.lastname*}*@hpi.uni-potsdam.de*
† *Operating Systems Group, Chemnitz University of Technology*
*peter.troeger@informatik.tu-chemnitz.de*

***Abstract*—Modern distributed systems have reached a level of complexity where software bugs and hardware failures are no longer exceptional, but a permanent operational threat. This holds especially for cloud infrastructures, which need to deliver resources to their customers under well-defined service-level agreements. Dependability need to be assessed carefully.**

**This article presents a structured approach for dependability stress testing in a cloud infrastructure. We automatically determine and inject the maximum amount of simultaneous non-fatal errors in different variations. This puts the existing resiliency mechanisms under heavy load, so that they are tested for their effectiveness in corner cases. The starting point is a failure space dependability model of the system. It includes the notion of fault tolerance dependencies, which encode fault-triggering relations between different software layers. From the model, our deterministic algorithm automatically derives fault injection campaigns that maximize dependability stress.**

**The article demonstrates the feasibility of the approach with an assessment of a fault tolerant OpenStack cloud infrastructure deployment.**

***Keywords*-fault injection, dependability modeling, testing, dependability stress, fault tolerance, OpenStack, fault tolerance dependency**

## I. Introduction

Modern cloud computing infrastructures need to have sophisticated mechanisms for fault tolerance and load balancing, so that varying user demands and unexpected events are considered appropriately and automatically. Given the complexity and rate of change in cloud infrastructure software, blind reliance on runtime mechanisms for achieving dependability is insufficient. Such systems need tailored testing procedures which must work automatically, so that scarce developer resources can be focused on feature development and bug fixing. Aside from the commonly known approaches for fault removal, namely the different flavours of unit testing, *software fault injection* (SFI) is a versatile tool for dependability assessment in failure space. SFI inserts – "injects" – different failure causes [1] into a running instance of the system. Observations and measurements of the system under this *fault load* indicate, whether the fault tolerance mechanisms work as expected.

Despite numerous SFI approaches from research [2], SFI is not yet established as a commonplace software development practice. This may be caused by usability issues among SFI tools or the challenge of answering the question of *when* and *where* to inject faults. The decision which software components to target through SFI is frequently made randomly or based on weak assumptions. This reduces the representativeness of the fault load [3].

We propose to re-evaluate the usefulness of SFI with the notion of *dependability stress testing*. It involves testing corner cases which are taxing for the implemented fault tolerance mechanisms, but still should be tolerated by the system and not lead to failure. Analogously to hardware stress testing, where single components are tested in physically extreme boundary conditions, we propose to evaluate software under the most severe fault load which, by specification, should still be tolerated.

Our approach of can be classified as an *integration testing* strategy. It comprises three steps: The first step consists of formulating a dependability model of the system (Section III). In the second step, a fault injection campaign, which maximizes dependability stress, is generated algorithmically based on this model (Section IV). Finally, this campaign is carried out on the system under test in an automated and orchestrated way. Relevant quality metrics are recorded and subsequently analyzed. If the campaign succeeds, this asserts that the system satisfies the dependability properties specified in the model. If it fails, detailed experiment results can help to identify and locate weak parts of the architecture.

The feasibility and potential of our approach is demonstrated at the example of an OpenStack setup (Section V).

## II. Related work

SFI can target various software layers, such as the operating system, a virtual runtime environment, or the application itself. Natella et al. provide a survey of such approaches [2].

FERRARI [4] uses software trap handlers to emulate CPU, memory, and bus faults. The tested process triggers the UNIX *ptrace()* facility to run in a special trace mode. Xception [5] relies on the advanced hardware capabilities of modern processors to achieve the same, using performance monitoring features to raise a processor debug exception when the injection condition is fulfilled.

0-7695-6330-9/17/31.00 ©2017 IEEE
DOI 10.1109/PDCAT.2017.00078



IEEE computer society

Fault injection recently became a prominent tool for evaluating distributed systems, e.g. by manipulating messages to inject faults into protocols [6], [7], or by creating distributed hardware fault injection environments [8], [9]. In [10], fault injection test cases for protocols are generated from *dependency trees*.

In cloud software systems, it is commonplace to inject single machine failures as faults in the overall distributed infrastructure. A widely known tool is *Chaos Monkey*, developed by Netflix. It randomly terminates *Amazon Elastic Compute Cloud* (EC2) instances. Netflix [11] and Amazon [12] use such tests for their production systems.

More recently, the term "dependability benchmarking" has been coined, describing the need for repeatable, uniform, and comparable ways of quantifying dependability aspects of complex software systems [13].

## III. Dependability modeling for fault injection

Empirical software evaluation needs to be based on an understanding of the expected system behavior. Due to the inherent and growing complexity of software [14], the formulation of adequate software failure cause models remains an open research topic [15]. Software dependability is generally threatened by hardware failures, software bugs, unforeseen interaction patterns, and non-determinism. Therefore, statements about the dependability need to be based on an understanding of the originating causes in the investigated system or its execution environment. Dependability models formalize this understanding.

Our approach works with the *fault tree* modeling language. Fault trees represent a systematic top-down deduction of failure causes starting with an undesired top event which denotes overall system failure. The leaf nodes of the tree, *basic events*, are connected hierarchically with Boolean logic gates until the undesired top event is reached. They thus constitute conceivable failure root causes. The failure-space view of fault trees matches the idea of SFI: Problematic events are artificially triggered to see if the system reacts as expected. Fault trees were originally defined in [16]. For software systems, one of the most important extensions was the addition of sequence-dependent gates [17].

If the system behavior under fault load and the fault tree model do not fit to each other, (at least) one of them must be flawed. This underlines the necessity to start with a trustworthy dependability model, f.e. by deriving it directly from architectural software descriptions [18]. In our use case – distributed cloud infrastructures – machines and network connections seem like a reasonable layer of abstraction for the dependability model. In subsequent sections, we assume that the system under test is modeled as a dynamic fault tree. Nevertheless, the proposed approach is not limited to this modeling language, but can be extended to other dependability model types.

## IV. Generation of SFI campaigns

Given the representation of the system as a fault tree, we now establish the notion of *dependability stress*. Dependability stress is a set of software failure causes which put load on the fault tolerance mechanisms of the system under investigation. With the absence or failure of fault tolerance mechanisms, this load would lead to a system failure. In analogy to hardware stress testing, the aim of software dependability stress testing should be to trigger corner cases, which are still tolerated, but may lead to significant performance impacts.

A single, controlled execution of the system with a certain amount of faults being injected is called *SFI experiment*. Each experiment creates a varying amount of dependability stress. A set of SFI experiments is called *campaign*. Campaigns can be designed according to different criteria. When maximizing dependability stress, we aim at **efficiency:** Since SFI imposes significant set-up, injection and tear-down overhead, the overall runtime of the campaign is a critical aspect. It should lie within a reasonable (application-specific) time frame. Our approach also focusses on **coverage:** For complex software systems, the space of possible failure cause combinations is vast. Nevertheless, the campaign should cover the most severe scenarios and exercise all fault tolerance mechanisms at least once. Finally, fault tolerance mechanisms should be tested at their limits and as **intensively**, as possible.

The efficiency of the campaign is influenced strongly by the number of experiments, since each experiment introduces mandatory set-up and tear-down effort. Therefore, the overall goal is to reduce the number of experiments within a campaign while still maintaining fault space coverage and maximizing dependability stress. The subsequently described approach results in an efficient, repeatable SFI campaign which maximizes dependability stress. As input, it relies on a dependability model. Although we discuss the approach at the example of fault trees, it is applicable to other modeling languages such as *reliability block diagram*s (RBDs) [19].

### *Step 1: Finding fault injection points*

We define a *fault injection point* (FIP) as an internal state change or external event which can contribute to fault activation, or directly causes a detectable error state. The goal of this step is to extract all FIPs from a given model. They are the smallest possible target for fault injection, thus reflecting granularity at which the system is modeled and tested. Extracting all FIPs from an existing fault tree model simply means listing all basic events.

For other dependability model types, similar approaches are conceivable: In RBDs, each unique block is an FIP. In Markov models, a state change from an error-free to an erroneous state is an FIP. Similarly, transitions from places representing correct states to places representing incorrect states can be regarded as FIPs in petri net models. In *failure*

*mode and effects analysis* (FMEA) tables, the table rows represent FIPs. The result of the FIP extraction step is a set of size $n$: $F = \{i_1, i_2, \ldots, i_n\}$. Depending on the type of system and the granularity level, at which fault injection takes place, $n$ denotes the number of physical, logical, or fine-grained software components, as well as additional failure causes such as external fault events.

*Step 2: Generating experiments*

An experiment $E$ is defined as a set of FIPs: $E \subseteq F$.

An experiment run represents exactly one program execution. Such an "execution" can be a period of normal system operation in an artificial or production environment, or a test case. The discussion of workload design is out of scope here; experiments focus on adding *fault load* to the system. An experiment is required to produce a binary, externally measurable outcome: It either succeeds, or it fails. We also distinguish between the set of *experiments which are anticipated to fail* $E_F$ and the set of *experiments which are anticipated to succeed* $E_S$. The generation of $E_F$ and $E_S$ can be based on the concept of *minimal cut set*s (mincuts), which are minimal unique combinations of basic events that can cause a system failure [16]. The mincuts can be extracted from a fault tree with established algorithms, such as *method for obtaining cut sets* (MOCUS) [20]. The experiments which are anticipated to fail $E_F$ correspond exactly to all experiments including a mincut. Inversely, any experiment set which does not include a mincut, must be assumed to succeed. Generating experiments from the dependability model ensures *repeatability*, which is an advantage over random fault injection approaches.

*Step 3: Maximal experiments*

Our approach assumes that experiments containing more FIPs put the system under more severe dependability stress than smaller experiments. To rank the experiments and choose the ones to be executed, we define the *maximality* criterion for experiments, which are anticipated to succeed ($E \in E_S$). An experiment is maximal, if it is impossible to add another FIP while still anticipating success:

$$max(E) \iff \forall E' \in E_S : E' \subseteq E \tag{1}$$

In other words, maximal experiments are ones which put the application under as much fault load as possible, while still assuming that the injected faults should be tolerable. The advantage of maximal experiments is that their success often implies the success of contained, smaller experiments.

Compared to naïve combinatorial testing, this approach significantly reduces test effort. It provides a reasonable balance regarding the trade-off between test effort and coverage: We only run a small subset of all possible experiments which nevertheless stresses the fault tolerance mechanisms as much as possible. Furthermore, we can guarantee coverage of all significant fault tolerance mechanisms being added by the developers. This tackles the commonly quoted problem of *fault load representativeness*: All experiments are based on the dependability model, which specifies "what should work". Maximal experiments have the potential to expose unforeseen synergistic effects. The significance of such effects and common-cause failures has been acknowledged by guiding standards in safety-critical industries [21].

Algorithm 1 creates a set of maximal experiments from a dependability model. It determines experiment candidates from all permutations of FIPs (the power set of $F$), by first checking whether they do not contain any mincut, meaning that are expected to succeed. As a result, we obtain $E_S$. The maximality criterion is applied to the experiment candidates in $E_S$, leaving only those who are not contained in any other member of $E_S$.

**Algorithm 1** Determining maximal experiments.

```
1:  function MaximalExperiments(model)
2:      F ← ObtainInjectionPoints(model)
3:      cutsets ← ObtainCutSets(model)
4:      A ← AllPermutations(F)
5:      E_S ← ∅
6:      for a ∈ A do
7:          containsCut ← a.ContainsAny(cutsets)
8:          if ¬containsCut then
9:              E_S ← E_S ∪ {a}
10:     E_S ← FilterFTDs(E_S)
11:     C ← ∅
12:     for i ∈ E_S do
13:         isSubset ← i.SubsetOfAny(E_S)
14:         if ¬isSubset then
15:             C ← C ∪ {i}
16:     return C
```

There are cases where the injection of maximal experiments prevents the fine-grained testing of error detection. When an event triggers an error state, which is detected and marked at a higher layer, this mechanism also needs to be tested, in which case, the experiment should not be maximal, but rather inject only the triggering event.

For example, the isolation of faulty nodes due to an internal problem renders the entire node unavailable. This is a more general error state at a higher level, which is intentionally triggered by the fault tolerance mechanism to avoid worse or less predictable outcomes. This is often denoted as *shoot the other node in the head* (STONITH) [22].

Since this scenario is very common in cloud-like distributed environments, we introduce the notion of a *Fault tolerance dependency (FTD)*, which describes a dependency between two FIP on different granularity levels. The "trigger" is a lower level fault event, which for the sake of error containment and isolation should cause the higher level "dependent". To test the detection and isolation mechanisms, experiments should never include the dependent if the trigger is also injected. Therefore, our Algorithm 1 contains a filter step which removes all experiments containing both the trigger and the dependent.

*Step 4: Deduction of campaigns*

Based on the maximal experiments, we now construct a *campaign* $C$ consisting of them. We obtain:

$$C = \{E_1, E_2, ...E_N\} \quad (2)$$

Each experiment is independent of other experiments, and the ordering of experiments does not matter: We assume, and have implemented in our example scenario, that the system is restored to the fully functional initial state after each experiment has run through. Despite the dependability stress, the system under test should continue its correct operation.

The campaign should be carried out in an orchestrated fashion and with proper logging of intermediate state and results, so its outcomes can be analyzed in-depth. The following section shows how this can be done in practice.

## V. Case study: Dependability stress testing of OpenStack

OpenStack is a popular open source cloud computing software stack written in Python. Private and industrial users run their own cloud environments based on the framework, expecting the same robustness and resiliency as with public clouds. For our case study of automated dependability stress testing, OpenStack is a suitable example: It is a complex real-world software system with strict dependability demands. Fault tolerance is a key aspect of OpenStack deployments. Due to its relatively short release cycles of six months, manual integration testing of OpenStack distributions can become cumbersome. OpenStack is composed of three major components: Compute ("Nova"), networking ("Neutron"), and storage, on which we focus.

Applying the approach described in the sections above, we built a tool chain for dependability stress testing. The core component is written in Python and allows for full end-to-end automation of the entire process of 1) analyzing the dependability model to establish candidate experiments in $E_S$, 2) generating an SFI campaign, which maximizes dependability stress, 3) coordinating the execution of the resulting campaign in a controlled environment. For the orchestration of the infrastructure components, injection of faults and collection of results, the IT automation tool Ansible[1] was employed.

### A. Test environment

Our OpenStack deployment runs on an HP Moonshot server system[2]. Multiple server cartridges, containing Intel Xeon E3-1284L processors and 32GB of RAM, are used as dedicated physical hosts, running Ubuntu 16.04 LTS.

The software setup relies on the *Fuel*[3] OpenStack distribution. We chose a configuration with two compute nodes, three storage nodes and three controller nodes. There is an additional `fuel-master` node which only plays a role during setup and is not needed for operation. Each node is a dedicated virtual machine (*Fuel VM*), configured as shown in Table I. Multiple nodes are combined on the same physical host and connected with VLANs, as shown in Figure 1.

Table I
Configuration of Fuel VMs.

| Node | Memory (GB) | Cores | Disc space (GB) |
|---|---|---|---|
| fuel-master | 28 | 6 | 50 |
| controller (3 nodes) | 14 | 4 | 64 |
| storage (3 nodes) | 14 | 3 | 40 |
| compute (2 nodes) | 14 | 4 | 28 |

The deployed storage backend is *Ceph*[4], a fault tolerant distributed block and object storage built upon the *RADOS* distributed storage system. The shared database service, which stores all relevant meta-data and runs on the controller nodes, relies on a MySQL Galera high-availability cluster. It tolerates at most one node outage.

On top of this OpenStack deployment, the distributed file store *Tahoe-LAFS*[5] serves as example application. Tahoe-LAFS is a decentralized storage system with client-side encryption designed around strict security concerns such as the principle of least privilege. In our configuration, Tahoe-LAFS needs all storage nodes (VMs running on OpenStack) to read or to write data. By having its redundancy features turned off, we ensure that failures of the underlying infrastructure (OpenStack) escalate and cannot be overlooked. For the performance evaluation, Tahoe-LAFS' internal performance counters are logged after a file has been written.

### B. Injection of faults

In the case study, we focus on the granularity of single virtual machines; we exclude low-level hardware faults and do not simulate application-level software faults. Instead, we focus on the dependability of the cloud management stack. All failure causes are injected at the same time in an experiment. While we assume that a maximal amount of simultaneously injected failure causes challenges fault tolerance mechanisms most intensely, our approach can be extended with different injection sequences.

Two types of failure causes were implemented and injected. The first type is *fail-stop behavior*, which denotes, that a component either works correctly or detectably crashes. The crash of a node was achieved by terminating a Fuel VM (i.e., "pulling the power cord") and by removing the link of a virtual network interface (i.e., "pulling the network cable"), respectively. Second, *fail-silent behavior* was implemented: A component either works correctly or does not respond. Such failure causes were injected by

[1]https://www.ansible.com/, November 4, 2017

[2]https://www.hpe.com/h20195/v2/getpdf.aspx/c04760473.pdf, June 2017

[3]https://docs.mirantis.com/openstack/fuel/fuel-9.1/mos-planning-guide/fuel-ref-arch/fuel_ref_arch_nodes_roles.html, November 4, 2017

[4]http://docs.ceph.com/docs/master/, November 4, 2017

[5]https://tahoe-lafs.org/trac/tahoe-lafs, November 4, 2017

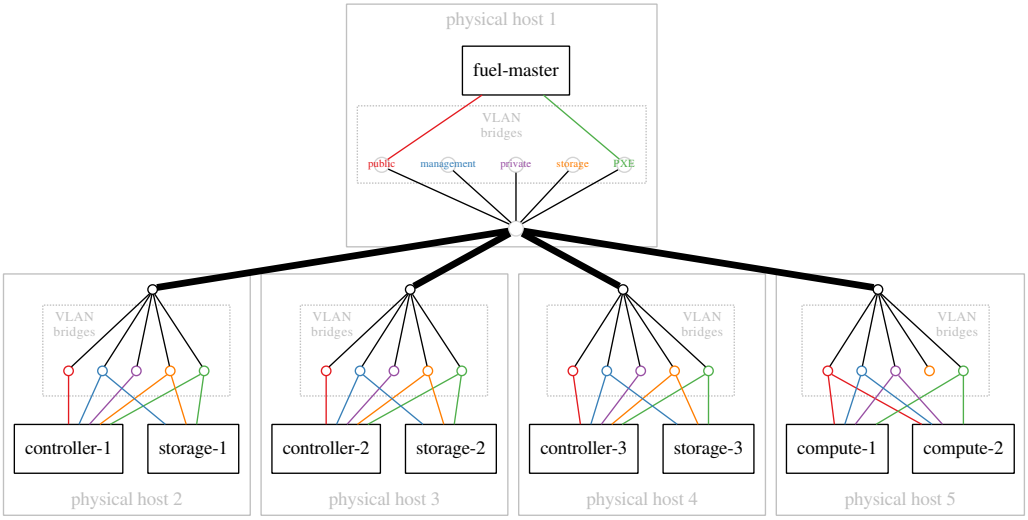


Figure 1. Topology of the OpenStack system. Bold links represent the physical network, colored links represent VLAN networks.

freezing a Fuel VM and by silently dropping outgoing network packets, respectively. Node faults are caused by scripting Fuel VM termination (fail-stop) or freezing (fail-silent); network faults are implemented using the *ebtables*[6] utility.

As first step, our implementation parses the given fault tree description, analyzes it as described in Section IV and generates campaigns with maximal experiments. The dependability model is created with the FuzzEd tool[7]. It generates machine-readible GraphML files representing the model.

The second step is the orchestrated execution of generated campaigns. Here, our implementation relies on exchangeable executables which know how to inject particular fault types and how to prepare for it. These executables are only identified by specific directory structures and can therefore be written in arbitrary ways: as Ansible Playbook calling VM management functions, as Python script using remote APIs, as Bash script using operating system functionality, or anything else.

Before each campaign enactment, the logging facilities are set up and a snapshot of each Fuel VM is created. These snapshots are used during the campaign to always return to a stable state before the next experiment. Experiments contain different stages. During the **pre-experiment stage**, snapshots of the Fuel VMs are restored on all nodes. Afterwards, the implementation waits for Tahoe-LAFS (our test application) to be accessible. Finally, a file containing random data is uploaded to Tahoe-LAFS. Next comes the **fault injection stage**: Based on the given failure cause model for the run, nodes are stopped, or network links are disconnected. In the final **post-experiment stage**, the uploaded data is read back and verified. If it proves to be of integrity, the result of the experiment is "success", otherwise "failure".

[6]http://ebtables.netfilter.org/, November 4, 2017
[7]https://fuzzed.org/, November 4, 2017

### C. OpenStack dependability model and campaign execution

As discussed in Section III, creating a dependability model is the first essential step to get suited fault injection campaigns. Figure 2 shows the fault tree for our setup.

From the application's point of view, the example system fails when data cannot be accessed correctly. This kind of top event is detectable, so it truly corresponds to an externally visible failure. It contains the assumption that the underlying OpenStack deployment mainly serves the purpose of offering dependable storage. The two compute nodes are not configured in a fault-tolerant setup in the given OpenStack configuration. For that reason, each compute node failure is a mincut. The system fails if the OpenStack compute service (Nova) or the block storage service (Cinder) fails. The latter can be caused by a failure of the object storage daemons (OSDs) themselves, or by a failure of the monitoring services. The latter run on the three redundant controller nodes, which tolerate at most one node failure. This is represented by the k-out-of-n gate with $k = 2$ on the fault tree's right hand side.

The Ceph OSDs tolerate the failure of up to two storage nodes – a k-out-of-n configuration with $k = 3$, because fault detection works differently here: Heartbeats are sent over a separate storage network. Unresponsive nodes are reported to a monitors running on the controller nodes and isolated. If the network itself fails, the node connected to it appears as failed to the heartbeat mechanism and is also isolated. We represent this mechanism, which we wish to test explicitly,

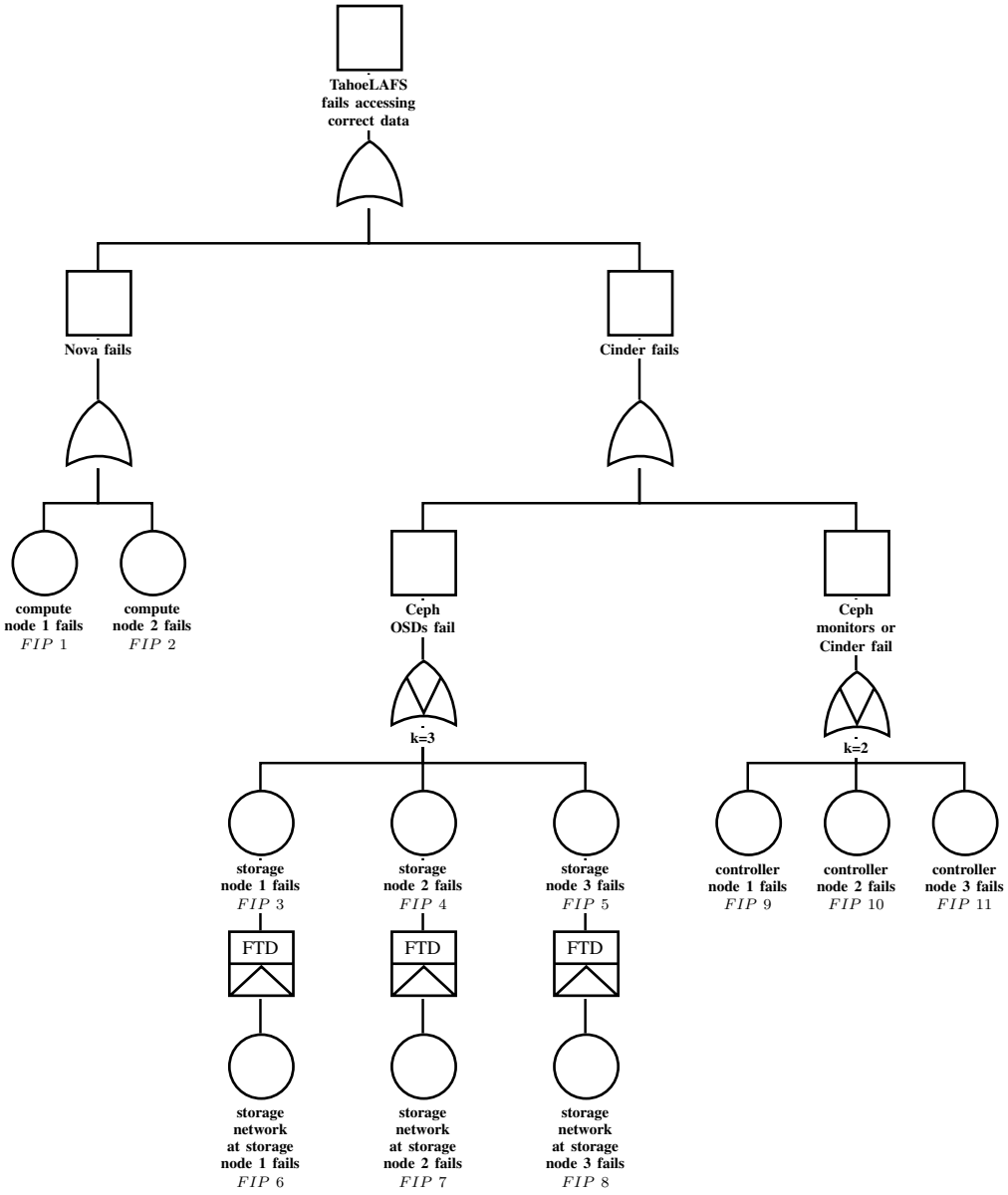


Figure 2. Dependability model of the test system. The fault tree depicts the root causes (basic events) which can lead to a failure to access correct data, as well as the fault tolerance mechanisms and redundancy installed to prevent such a failure.

as a *fault tolerance dependency* (FTD) gate. The notation is based on the *FDEP* gate from traditional fault trees, which also has an input trigger event and several dependent events.

The dependability model contains 11 basic events, hence 11 FIPs. With a naïve combinatorial approach, injecting all conceivable permutations of FIPs would amount to $2^{11} = 2048$ different experiments. When we applied our algorithm to this model, we obtained 36 maximal experiments.

These experiments are listed in Table II. The table shows the FIPs which were identified, along with the information which of them are injected per experiment. $FIP$ 1 and $FIP$ 2 are never injected – each of them constitutes a single point of failure, so they are not expected to be tolerable and therefore do not contribute as dependability stress scenario. The FTD filtering leads to the fact that $FIP$ 3 is not injected when $FIP$ 6 is already being injected. The same argumentation holds for $FIP$ 4 / $FIP$ 7 as well as $FIP$ 5 / $FIP$ 8. Thus, FTD filtering ensures that the detection of missing heartbeats is checked as part of the experiments.

On average, the campaign takes around five hours if each experiment is carried out once. The average experiment runtime is 573 seconds, of which the majority (over 500

Table II
EXPERIMENT CONFIGURATIONS IN THE FAULT INJECTION CAMPAIGN.

| ID | FIP 1 | FIP 2 | FIP 3 | FIP 4 | FIP 5 | FIP 6 | FIP 7 | FIP 8 | FIP 9 | FIP 10 | FIP 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| af17 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| e8d5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| bfa5 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| cf08 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| a5df | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 9d46 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 2815 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| cff5 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| bbad | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 4a31 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| ae5d | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 027b | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| fcf8 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0974 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1388 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1b0f | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 883a | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 721b | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 2314 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 3ef2 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| a78b | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| ade4 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 3e53 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| d784 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| cff3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| cf40 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 16b5 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 4c28 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 9498 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 73c9 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 06fa | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 60d1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| e313 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| fb1a | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 36fa | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| c60e | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

seconds) is spent with environment set-up prior to the actual fault injection. These numbers underline that naïvely running all experiments is impractical in real-world systems.

We carried out each experiment ten times to gain a statistical confidence. The resulting performance degradation under fault load is plotted in Table III. The numbers represent the ratio between the time it takes to upload and store the random data in Tahoe-LAFS under fault load and in fault-free conditions.

The overall system did not crash or freeze in any of our experiments. In all cases, writing data to Tahoe-LAFS was possible without irrecoverable failure, although the performance was impacted negatively by the fault load. As Table III shows, the severity of performance degradation varies considerably for different experiments and failure cause models.

For example, the experiment with ID *bbad* has a big impact on performance both in the fail-stop and fail-silent cases. It comprises the injection of $FIP$ 3 ("storage node 1 fails"), $FIP$ 8 ("storage network at storage node 3 fails"), and $FIP$ 11 ("controller node 3 fails"). The MySQL cluster uses heartbeats to detect errors, which may cause additional delays.

Our approach provides detailed observations related to single experiments, traceable in the dependability model, which can be starting points for further detailed investigations. We have introduced an implementation which eases and automates carrying out identified critical experiments

Table III
AVERAGE PERFORMANCE DEGRADATION (10 RUNS PER EXPERIMENT)

| Experiment ID | Fail-stop | Variance | Fail-silent | Variance |
|---|---|---|---|---|
| af17 | 2.05 | 0.84 | 2.20 | 0.79 |
| e8d5 | 2.24 | 0.74 | 2.21 | 0.74 |
| bfa5 | 2.04 | 0.86 | 1.99 | 0.87 |
| cf08 | 2.04 | 0.83 | 1.71 | 0.74 |
| a5df | 2.01 | 1.26 | 2.63 | 1.18 |
| 9d46 | 2.04 | 0.93 | 2.07 | 1.17 |
| 2815 | 2.04 | 1.22 | 1.79 | 1.42 |
| cff5 | 1.65 | 1.00 | 1.22 | 0.45 |
| bbad | 2.64 | 0.26 | 2.39 | 0.68 |
| 4a31 | 2.18 | 0.89 | 1.79 | 0.96 |
| ae5d | 1.87 | 0.77 | 1.50 | 0.64 |
| 027b | 1.66 | 0.82 | 2.06 | 1.08 |
| fcf8 | 1.83 | 0.92 | 1.23 | 0.63 |
| 0974 | 2.48 | 0.94 | 2.41 | 0.78 |
| 1388 | 1.47 | 0.97 | 1.82 | 1.56 |
| 1b0f | 1.52 | 1.18 | 2.04 | 0.67 |
| 883a | 2.20 | 0.72 | 1.81 | 0.97 |
| 721b | 1.96 | 0.79 | 1.51 | 1.02 |
| 2314 | 2.13 | 1.55 | 2.11 | 1.30 |
| 3ef2 | 2.11 | 0.56 | 1.80 | 1.02 |
| a78b | 1.73 | 0.47 | 2.18 | 1.34 |
| ade4 | 1.27 | 0.57 | 1.66 | 0.68 |
| 3e53 | 2.48 | 1.29 | 2.00 | 0.68 |
| d784 | 1.33 | 0.75 | 2.00 | 1.76 |
| cff3 | 1.57 | 0.56 | 1.46 | 0.55 |
| cf40 | 2.36 | 0.66 | 1.73 | 0.91 |
| 16b5 | 1.36 | 0.53 | 2.29 | 0.86 |
| 4c28 | 2.08 | 0.90 | 2.23 | 0.60 |
| 9498 | 1.88 | 1.04 | 2.37 | 0.82 |
| 73c9 | 2.06 | 0.94 | 1.97 | 1.14 |
| 06fa | 1.65 | 0.72 | 2.03 | 1.12 |
| 60d1 | 1.78 | 0.66 | 2.27 | 1.15 |
| e313 | 1.43 | 0.43 | 2.53 | 0.90 |
| fb1a | 1.48 | 0.56 | 1.30 | 0.29 |
| 36fa | 2.12 | 0.68 | 2.04 | 0.97 |
| c60e | 1.14 | 0.38 | 1.60 | 0.90 |
| **Average** | **1.89** | | **1.94** | |

(such as *bbad*). This can enable a deeper integration of SFI testing into the software development process.

## VI. Conclusion

Our approach enables repeatable and automated dependability assessments, especially suited to distributed software systems. The presented algorithm derives a SFI campaign from a dependability model, and is capable of taking fault tolerance dependencies (FTDs) into consideration. It tests for synergistic effects by relying on the concept of *dependability stress*, which describes maximum non-fatal sets of failure root causes that are simultaneously injected.

The feasibility and real-world applicability of the approach was demonstrated on the case study of an fault tolerant OpenStack deployment. In comparison with existing SFI strategies, we see the capability for scalability and automation as a major benefit of our approach. With only a small subset of all conceivable experiments, statements about the "worst case" performance in the presence of a fault load are possible.

Our campaign optimized for dependability stress contains the most critical sets of failure causes, which can be injected in a repeatable and observable manner. Thus, the approach also constitutes progress towards a distributed debugging environment for OpenStack and similar deployments.

An obvious next step is the ranking of fault injection points according to their contribution to performance degradation. Experiments could also be extended by the notion of timing or sequence-dependencies. Continuous integration, running upon each software modification, can be extended to accomodate SFI using the approach presented here. For this purpose, machine-readable infrastructure information is increasingly available with the emerging *Infrastructure as Code* trend.

## References


[1] P. Tröger, L. Feinbube, and M. Werner, "What activates a bug? A refinement of the Laprie terminology model," in *26th IEEE International Symposium on Software Reliability Engineering*, 2015.

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

[3] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, 2013.

[4] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, Feb 1995.

[5] J. Carreira, H. Madeira, and J. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *Software Engineering, IEEE Transactions on*, vol. 24, no. 2, pp. 125–136, 1998.

[6] S. Dawson, F. Jahanian, and T. Mitton, "Orchestra: A fault injection environment for distributed systems," 1996.

[7] N. Looker and J. Xu, "Dependability assessment of grid middleware," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. IEEE, 2007, pp. 125–130.

[8] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*. IEEE, 2000.

[9] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "FIAT-fault injection based automated testing environment," in *Fault-Tolerant Computing, 1988. FTCS-18, Eighteenth International Symposium on*, June 1988.

[10] P. Sinha and N. Suri, "Identification of test cases using a formal approach," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*. IEEE, 1999, pp. 314–321.

[11] A. Tseitlin, "The antifragile organization," *Communications of the ACM*, vol. 56, no. 8, pp. 40–44, 2013.

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

[13] H. Madeira and P. Koopman, "Dependability benchmarking: making choices in an n-dimensional problem space," 2001.

[14] B. Beizer, "Software is different," *Annals of Software Engineering*, vol. 10, no. 1, pp. 293–310, 2000.

[15] L. Feinbube, P. Tröger, and A. Polze, "The landscape of software failure cause models," *arXiv:1603.04335*, 2016.

[16] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," 1981.

[17] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Fault trees and sequence dependencies," in *Reliability and Maintainability Symposium, 1990. Proceedings., Annual*. IEEE, 1990.

[18] C. Lauer, R. German, and J. Pollmer, "Fault tree synthesis from uml models for reliability analysis at early design stages," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 1, pp. 1–8, 2011.

[19] M. Malhotra and K. S. Trivedi, "Power-hierarchy of dependability-model types," *IEEE Transactions on Reliability*, vol. 43, pp. 493–502, 1994.

[20] J. Fussell, E. Henry, and N. Marshall, "MOCUS: a computer program to obtain minimal sets from fault trees," 1974.

[21] "ISO 26262:2011(E) Road vehicles – Functional safety," Nov 2011.

[22] A. Robertson, "Resource fencing using STONITH," *White Paper, August*, 2001.