This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems

Failure Diagnosis for Distributed Systems using Targeted Fault Injection

Cuong Pham*, Long Wang+, Byung Chul Tak+, Salman Baset+, Chunqiang Tang†,
Zbigniew Kalbarczyk*, Ravishankar K. Iyer*,
* University of Illinois at Urbana-Champaign + IBM Thomas J. Watson Research Center † Facebook Inc.

Abstract—This paper introduces a novel approach to automating failure diagnostics in distributed systems by combining fault injection and data analytics. We use fault injection to populate the database of failures for a target distributed system. When a failure is reported from production environment, the database is queried to find "matched" failures generated by fault injections. Relying on the assumption that similar faults generate similar failures, we use information from the matched failures as hints to locate the actual root cause of the reported failures. In order to implement this approach, we introduce techniques for (i) reconstructing end-to-end execution flows of distributed software components, (ii) computing the similarity of the reconstructed flows, and (iii) performing precise fault injection at pre-specified executing points in distributed systems. We have evaluated our approach using an OpenStack cloud platform, a popular cloud infrastructure management system. Our experimental results showed that this approach is effective in determining the root causes, e.g., fault types and affected components, for 71-100% of tested failures. Furthermore, it can provide fault locations close to actual ones and can easily be used to find and fix actual root causes. We have also validated this technique by localizing real bugs that occurred in OpenStack.

1 Introduction

Software errors are one of the major threats to the availability of production systems [26], [37]. Even with many advances in software testing, software bugs in distributed systems continue to escape and impact critical services. This is mainly due to the high cost of testing, and the separation of testing and production environments. In addition, software is hardly a stationary target. A newly implemented feature could introduce a new error, and even an error fix could introduce another error [23].

Once a production system is impacted by such an error, it needs to be quickly remediated, i.e., diagnosed and then fixed to minimize system unavailability which may lead to serious financial losses. Diagnosis refers to the identification of a failure's root cause (e.g., a software error). This process usually involves meticulous analysis of execution traces (e.g., from log files) and inspection of system's faulty executions (e.g., with the assistance from debugging tools). This process is laborious and as we show is well-suited for computer automation. Various approaches have been studied in the domain of automated software failure diagnosis for distributed systems. Methods such as static [39] and dynamic [34] program slicing analyze the state of only single-process applications. Spectrum-based techniques [1] deal with only the internal process events, such as the number of predicate hits [27] or the number of function call sequence hits [11], [29]. In order to support failure diagnosis of distributed systems, it is essential to be able to correlate events generated across components and perform analysis on the entire event flow.

This paper introduces a novel approach to automating failure diagnosis of distributed systems. Our approach is based on the combination of a new fault injection (FI) technique and data analytics. FI is a well-established technique for assessing the robustness of software [19]. It involves inserting artificially generated errors, such as premature process termination, memory data corruption and/or mutilation of function return values. Our fault injection technique departs from existing approaches in that we have a capability to control the injection locations by leveraging the knowledge of execution flows across distributed components. In our case, execution flow is the sequence of system calls or library calls (spanning multiple components) invoked during the processing of user requests. We have developed a mechanism to track flows and to inject faults at arbitrary points along the flows. This mechanism allowed us to systematically cover the fault spaces during fault injection campaigns.

The outcome of our research is the tool, called *Targeted Fault Injection (TFI)*. It consists of several functionalities, including *(i)* aforementioned mechanism for injecting faults at specified points in the execution flows, *(ii) FI Specification Language* that allows users to designate various configuration parameters of fault injections during the runtime, *(iii)* control modules for overseeing the entire fault injection campaigns, and *(iv)* components for storing specifications, fault injection states and the failure profiles collected from the FI runs into the Failure Profile Database (FPDB). To diagnose a newly submitted failure, its execution flow trace is first compared against the execution flow traces stored in the FPDB using our custom similarity metrics. Then, a ranked list of candidate faults are composed from the faults that generated those similar execution traces. From our experiments, we find that obtained fault information, such as fault type (e.g., process crash or message corruption) and the fault location (e.g., a location in an execution trace), is a good indicator for identifying potential root causes of the given failure.

Comparing execution flows of distributed execution has challenges since they can branch out to parallel executions or merge into one, forming complex graph. To quantify the similarity of two failures, we have developed a method to causally order events that are recorded across distributed processes into a single

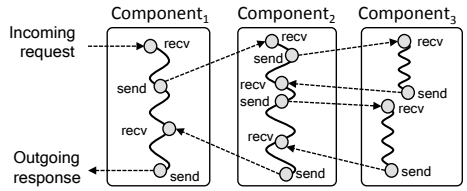


Fig. 1. An example processing flow consisting of *send*() and *recv*() LibC calls across distributed components. A full processing flow stored in our database also includes other types of LibC calls, which are ordered based on the causal relationship of the sending and receiving events.

event sequence, or *processing flow*. [1]. Once execution flows are serialized, we define *the distance between two ordered event sequences* as the smallest number of events that need to be added or removed in order to transform one sequence into the other (i.e. string edit distance).

We have evaluated the proposed failure diagnosis method, *TFI*, on OpenStack [31], a widely used distributed cloud management system. We have verified that our proposed framework could accurately determine the failure types and the affected components for 71-100% of tested failure cases. Furthermore, the returned fault locations were close to the locations of actual faults, and provided good indication of actual fault-to-failure propagation paths. We have used examples of real bugs reported in OpenStack to demonstrate the diagnostic capabilities of our approach.

The rest of the paper is organized as follows. Section 2 describes the design of our proposed framework and its implementation requirement. Next, Section 3.3.1 presents the implementation of the Targeted Fault Injection tool, which is the core technology of the framework. After that, Section 5 presents our evaluation with OpenStack. Section 6 reviews relevant literature to our work. Finally, Section 7 concludes the paper.

2 Framework Overview

2.1 Failure Diagnosis Workflow

Our failure diagnosis workflow is as follows. We first collect traces from production systems during executions that led to failures. That task is done by instrumenting production systems with a distributed tracing tool. Our tracing mechanism is conceptually similar to vPath [38], Dapper [35], and Magpie [7] for collecting traces of large-scale production distributed systems. In case the tracing tool is not available in a target production system, a failure must be reproduced in a separate testing environment with the tracing tool enabled to obtain a trace of the failure.

After that, the collected traces are used to reconstruct a processing flow corresponding to the failure. A processing flow is a sequence of causally ordered system events (e.g., LibC calls or API calls) invoked across multiple components during the processing of a request. A sequence begins at the first event indicating that an external request (e.g., an HTTP request from a client) has been received by the system, and ends at the last event indicating that a response to the request has been returned or the processing of the request has terminated (e.g., because of a failure). An example processing flow of a request is given in Figure 1; the collected events include *send*() and *recv*() LibC function calls invoked across three components.

1. An *execution trace*, or *trace*, is a *raw* record of collected events. A *processing flow*, or *flow*, is an *ordered* sequence of events that follows their causal relationships.

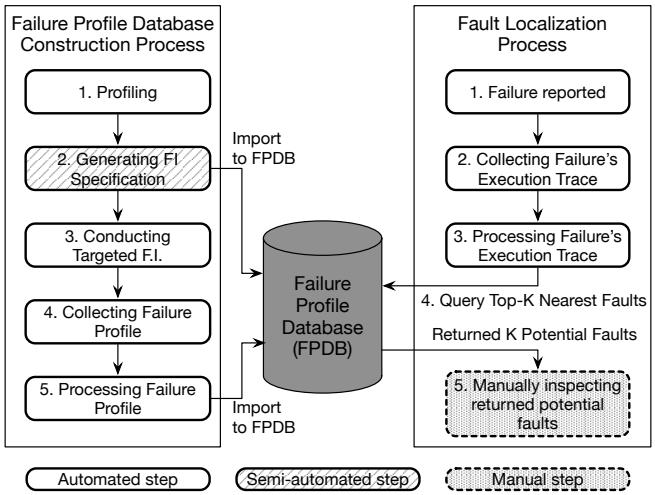


Fig. 2. Overview of the processes in failure diagnosis framework using Targeted Fault Injection.

A reconstructed processing flow is then used to query against a preconstructed Failure Profile Database (FPDB) to find faults that generate the same or similar processing flows. The identified faults (and their locations in the processing flows) are given to developers as hints for determining the actual root causes of the observed failures.

Figure 2 depicts the process in the proposed framework. There are two phases in the failure diagnosis process: (i) *Failure Profile Database construction* and (ii) *fault localization*. The following sections describe each phase in more detail.

2.2 Failure Profile Database Construction

FPDB construction is the process of collecting, processing, and storing failure profiles and relevant data about root causes of observed failures. In previous studies [12], [15], [28], concepts similar to that of FPDB was proposed, wherein the data of *known* failures are used to construct a database. A major limitation of all these studies is while the provide a view of past failure, they cannot predict locations of new failures. We overcome that limitation by using fault injection to populate an FPDB. In our FPDB, each fault injection experiment generates a failure profile entry, which consists of *a description of the injected fault* and *a set of execution traces* collected by repeated injections of the same fault in several independent runs. We collect multiple execution traces generated by the same fault to account for variations in the processing flow of each fault. A variation might be caused by nondeterminisms in execution, which can only be learned by observing multiple executions while applying the same perturbation (i.e., fault). The FPDB is constructed through the following five steps:

Step 1: Profiling. This is a prerequisite for setting up a fault injection experiment. We employ the technique discussed in Section 3.1 to reconstruct processing flows from failure-free executions of request types that are to be included in the FPDB. The resulting flow information is used in subsequent steps to define fault injection experiments.

Step 2: Specifying fault injection experiment plan. The following parameters are used to specify a fault injection plan.

Injection Granularity defines the unit of computation at which a fault is triggered. An injection granularity directly affects the accuracy of the fault localization. For example, if the granularity

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems

is at the level of LibC calls, a fault is injected when there is a call to a LibC function. Thus, a fault localization using these data points can only indicate that the problem has occurred around the area of the given LibC call.

Trigger Location indicates an event (e.g., a LibC call) at which a fault is triggered. Given an injection granularity, one can enumerate all possible trigger locations along an end-to-end processing flow obtained in step 1. In our framework, each trigger location is described by a sequence of causal events that precedes the trigger event. An example of a sequence of causal events is: "after message M is sent from component A to component B, wait for 10 system call invocations in component B, and then inject a fault into component B". We can also specify value ranges as an event to cover a certain region of a processing flow. For example, instead of specifying exactly the 10th system call invocation, we can specify a range from the 10th to 20th system call invocations.

Fault Model defines what type of fault to inject. We choose three fault models: *process crash*, *deadlock*, and *message corruption*. They represent two broad classes of failure modes in distributed systems: contained failures and propagated failures. A *contained failure* is a failure that occurs in one process and does not propagate to others. Process crash and deadlock are common types of contained failures. Mechanisms for injecting crashes and deadlocks ensure that no propagation is possible before a process terminates or stops making further progress. Conversely, a *propagated failure* affects the application beyond process boundaries, often via distribution of corrupted messages.

Injection Target Location defines where to inject a fault after it is activated. A target can be a computing node (e.g., a virtual machine), a process, or a thread. Depending on the fault model, this attribute may include more specific information, such as which part of a message to corrupt.

Step 3: Conducting targeted fault injection. This step is responsible for executing the fault injection experiments as specified in step 2. We design and implement a fault injection tool, called TFI (Targeted Fault Injection), that dynamically tracks and correlates events generated by a distributed system at runtime to deterministically inject faults as instructed by the fault injection plan. For each injection specified in a plan, the TFI tool (i) constructs a state machine based on the sequence of causal events described by *Trigger Location* information of the specification; (ii) intercepts the corresponding events at runtime to trigger state transitions within the state machine; and then (iii) injects the specified fault to the specified target at the appropriate time.

Step 4: Failure profile collection. This step collects failure profiles generated by fault injection experiments. As mentioned above, a failure profile contains a fault specification and a set of corresponding traces.

Step 5: Processing failure profiles. This step restructures raw events in each trace into a processing flow to capture the causal relationships among events. Each processing flow is then converted into a string of characters, which enables the calculation of the similarity metric between two arbitrary traces. The converted flows and the specifications of the corresponding injected faults are finally stored in the FPDB.

2.3 Fault Localization

Fault localization refers to a process of narrowing down potential failure root causes to a small set of the most likely ones. Our framework does so by searching an FPDB for the injected faults that generated the most similar processing flows to the one associated with the new reported failure. When a failure is reported (step 1 in the fault localization process depicted in Figure 2), its trace is collected (step 2) and then processed (step 3). The trace-processing step (step 3) is the same as step 5 in the FPDB construction process.

The processing flow of the reported failure is used to query the FPDB to find potential root causes (step 4). The FPDB supports the *Find Top-K Nearest Faults* query by utilizing the *Execution Trace Distance* function $\delta(t1,t2)$. $t1$ and $t2$ denote two arbitrary traces. The underlining algorithm to compute an execution trace distances $\delta(t1,t2)$ is the *edit distance* [5], [30] (a.k.a. the Levenshtein distance) of two strings that represent the two traces $t1$ and $t2$. Section 3.2 details the methods to convert an execution trace to a string representation and compute the edit distances.

The *Find Top-K Nearest Faults* query takes a reference trace t_r and an integer value K as input, and then returns K faults in the FPDB that have the smallest distances to the reference trace t_r (see Algorithm 1). For example, if the input t_r is a trace of a new failure to be diagnosed, the returned faults provide information

```
Data: T_FPDB(f): Set of traces stored in FPDB which are
      generated by the same fault f
Data: F_FPDB: Set faults stored in FPDB
func DistanceToFault(t_r, f_r) {
    Input: t_r: reference trace; f_r: injected fault.
    Output: Distance from t_r to group of traces generated by
            fault f_r.
    T_db = T_FPDB(f);
    for i ← 1..n do
        δ_i = EditDistance(t_r, t_dbi);
    end
    σ = StandardDeviation({δ_1, .., δ_n});
    for i ← 1..n do
        Δ_i = e^(-δ_i^2 / 2σ^2);
    end
    return Σ_{i=1}^{n} Δ_i/n;
}
func TopKNearestFaults(t_r, K) {
    Input: t_r: reference trace; K: for Top-K.
    Output: K faults generated the most similar traces to t_r.
    TopKF = ∅;    // Top-K nearest faults
    TopKD = ∅;    // Top-K nearest distances
    for f ∈ F_FPDB do
        d_f = DistanceToFault (t_r, f);
        (d_max, f_max) ← MaxDistance(TopKD, TopKF);
        if size(TopKF) < K then
            TopKF = TopKF + {f};
            TopKD = TopKD + {d_f};
        else
            if d_f < d_max then
                TopKF = TopKF − {f_max} + {f};
                TopKD = TopKD − {d_max} + {d_f};
            end
        end
    end
    return TopKF;
}
```

Algorithm 1: Pseudo-code of the *Find Top-K Nearest Faults* query. Function `DistanceToFault` is for computing the distance from a reference trace (t_r) to a group of traces generated by the same fault f_r stored in the FPDB.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems

on the potential locations of the actual, thus unknown, fault that generated t_r.

The distance between the trace t_r and an injected fault f is the average *Gaussian influence* [17] between t_r and all traces t_{db} in the failure profile of fault f stored in the FPDB. Recall that each fault has multiple traces in a failure profile entry. The Gaussian influence is computed as follows:

$$\Delta_{Gaussian}(t_r, t_{db}) = e^{-\frac{\delta(t_r,t_{db})^2}{2\sigma^2}}$$

where σ is the *scaling factor*, which is the standard deviation of the pair-wise distances of all fault f's traces stored in the FPDB.

The use of Gaussian influences is intended to account for the variation of traces generated by the same fault. We observe that some faults tend to generate traces with higher variation (i.e., less deterministic) than others. We attribute that behavior to the nature of each fault. Therefore, when Gaussian influence is used to compute the distance of a trace to a group of traces generated by the same fault, the variation in the group of traces is used as the scaling factor: the computed distance is positively correlated with the variation in the trace group.

The function `DistanceToFault` in Algorithm 1 illustrates how Gaussian influence is used to compute the distance between a reference trace and a group of traces. The pseudo-code in Algorithm 1 illustrates the underlining procedure to execute the Find Top-K Nearest Faults query.

3 Enabling Techniques

Three key techniques constitute the core of the proposed diagnosis framework: (i) processing flow discovery, (ii) processing flow representation and comparison, and (iii) targeted fault injection. The following subsections describe each of them in detail.

3.1 Processing Flow Discovery

Successful execution of our method requires a tracing mechanism that can track message flow across distributed components. A flow of messages contains causal relationships of messages exchanged across distributed components. For instance, suppose that a component (e.g., an Apache web server) receive two HTTP requests, and later, the same component generates one SQL query to a MySQL database server. Which of those two HTTP requests is responsible for the generation of the SQL query message? Answering such questions requires discovery of the causal relationship between messages. One approach is to modify the application source code (i.e., Apache and MySQL in the example) so that every message contains a global identifier. Another way is to perform sophisticated inference on observable information, such as timing, to infer the causality without modifying applications. In order to make such an approach practical, our tracing is designed to be non-intrusive to applications while still producing a precise trace of processing flows.

The tracing mechanism employed in this work adopts the principles proposed in vPath [38]. The main goal of tracing is to track messages between components of a multi-threaded application. Once a message is received, it is assigned to one of the threads in the component, and that thread is responsible for the processing of the message until it generates another outgoing message. That method implies that, after the message is received, causality can be tracked by observing thread IDs up to the point where a thread sends out a new message. These principles can be realized by monitoring system calls related to network activities, such as *send*() and *recv*(). We record the thread ID of the caller thread and the socket tuple information (local IP address, port) and (remote IP address, port) upon detecting those system calls. The thread ID is used to tie events from *recv*() to *send*() (i.e., within components), and the socket information to tie events from *send*() to *revc*() (i.e., across components).

However, tracking messages across multi-threaded components is insufficient. For scalability, many modern applications employ asynchronous message queues such as AMQP [3]. Since messages are inserted by one thread and picked up by another at a later time, the causality within the queue is not carried by threads, but rather by the messages themselves. In order to enable message tracking across queues, we need to look at the message contents to recognize the message identifiers, and correlate incoming and outgoing messages.

We have implemented those principles within a custom shared library. It contains a set of predefined library functions that perform network I/O, such as *recv*(), *recvfrom*(), *send*() or *sendto*(), and other system calls. We interpose this custom shared library in front of the LibC library using the *LD_PRELOAD* mechanism [24]. This way, we are able to intercept LibC calls related to network activities and record appropriate data. The collected data from all the processes in the target system constitute the raw trace of an execution.

3.2 Failure Profile Representation and Comparison

After collecting a raw trace, it needs to be transformed into a format that allows quantification of the similarity between a pair of traces. Specifically, we transform each collected trace into a single string, which represents (i) messages exchanged between components, and (ii) the number of system events (e.g., LibC calls) that a component invokes for processing an incoming message.

3.2.1 Data Cleanup

Automated diagnosis using FPDBs assumes that request processing is deterministic and events in the causal sequence of request processing maintain a deterministic ordering (the events may be concurrent and still deterministic). Thus, collected traces must be pruned to remove nondeterministic events, such as system noise, message fragmentation, and out-of-order messages.

System Event Noise. It is common for an application to use periodic messages, such as timeout or heartbeat messages, to update the status (e.g., the liveness) of distributed components. Periodic messages do not belong to any processing flows of user requests. However, they are captured in our traces, and their order relative to other messages is nondeterministic.

In order to remove those messages, we developed a heuristic-based algorithm to detect periodic patterns of messages and timestamps in a trace. The algorithm works on a set of events generated by a component. First, the algorithm searches for *the most frequent* group of consecutive messages; then it determines whether the occurrences of these groups are *periodic in time*. Specifically, the first step of the algorithm generates n-gram sequences of a flow (where, n is the number of messages in each group, and the value of n is incrementally increased until no more new patterns are found), and then computes a hash table of the generated sequences. The second step of the algorithm processes the sequences in the largest buckets of the hash table. For each sequence in a bucket, the time interval between the two members of each pair of consecutive

sequences is computed using the timestamp of the messages. A bucket of messages is determined to be periodic if the variation in the time intervals between arrivals of two subsequent messages is smaller than a certain threshold.

Fragmented Messages. Because of the implementation of the TCP (Transmission Control Protocol) stack, a message may be fragmented unpredictably. That fragmentation is also non-deterministic across runs. We group events to eliminate the non-determinism caused by fragmented messages. For example, consecutive *send*() or *recv*() events that have identical source and destination ports are grouped into one event, as the separate message are a result of fragmentation.

Non-deterministic Message Ordering. Because of network latencies, distributed data collection often suffers from unordered messages. We find all events that violate causal orderings and reorder them. For example, consider a pair of *send*() and *recv*() events, such that the *recv*() arrives at the centralized logger before the *send*(); the order of these event needs to be switched.

3.2.2 Organizing Concurrent Processing

In a distributed system, a task can be split into multiple concurrent sub-tasks. For example, a main component distributes messages to multiple worker components in an orderly manner to perform simultaneous computation. The main component then gathers the results from messages sent by the workers after they finish their sub-tasks. The order in which those completion messages are received depends on the performance of each worker, which is largely nondeterministic.

To remove that nondeterminism of gathered messages, we represent the causal relationships of *sending* and *receiving* events in a tree structure, in which:

- a *node* is a sending or receiving event,
- the *root* of a tree is the first event in the trace,
- the *parent of a non-root sending event* is the preceding event in the same thread (e.g., determined by Thread IDs),
- the *parent of a non-root receiving event* is the corresponding sending event, from which the received message was transmitted.

That tree structure captures the causal relationship among receiving and sending events rather than their timing information. Therefore, the order in which messages are received across multiple components does not affect the tree structure.

We need to complete the tree to include information about the internal execution within each thread. After the previous step, sending and receiving events form a causal relationship tree, which provides a skeleton of the entire processing flow. We add details to the skeleton by inserting other events, e.g., other LibC calls, between consecutive sending and receiving events in the same thread. The tree now contains all the events captured during the execution of a request.

3.2.3 Encoding Failure Profiles

We convert the tree structure obtained from the previous step into a single string of characters. Each process in the target distributed system is mapped to a unique *character* (note that we can use Unicode, which allows us to represent a large number of processes). Then, each node in a tree is labeled with the character that represents the process that triggered the event. Finally, to obtain a representation for a tree, we traverse the tree in a specific order (e.g., post-order) and record the sequence of characters in the labels of the nodes being visited along the traversal. That entire sequence of characters is stored as a string in the FPDB.

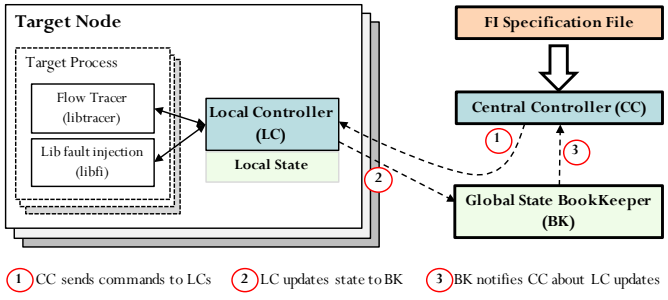


Fig. 3. Overall architecture of the Targeted Fault Injection framework.

3.2.4 Computing Trace Distances

We use the *string edit distance* metric [5], [30] to compute the Execution Trace Distance δ between two traces, which are represented as two strings using the method presented above. String edit distance is defined as the minimum number of *insertions, deletions, and replacements* of characters required transforming one string into another. For example, the strings "hello world" and "hey world!" have an edit distance of four (replacement of the first "l" with "y", deletion of the second "l" and the following "o", and insertion of "!").

The more similar the two traces are, the smaller their distance δ is. Using that metric, finding the most similar trace to a reference trace t_r is equivalent to finding the trace that has the smallest distance to t_r. Computation of the edit distance metric is used to compute the Gaussian influence used in the Top-K query (Section 2.3).

3.3 Targeted Fault Injection (TFI)

The proposed FPDB construction method (see Section 2.2) requires that every fault be activated at a precise predetermined location, i.e., the trigger location. That capability is key (i) to construct an FPDB that covers all possible locations specified by an *injection granularity*, and (ii) to accurately interpret information on fault locations returned by the FPDB to the user. We developed a tool called *Targeted Fault Injection* (*TFI*) to carry out that task.

3.3.1 Design of TFI

TFI injects a fault immediately preceding the last event in a causally ordered sequence of events. For example, suppose that TFI is instructed to inject a fault f at the end of the event sequence $e_1 \rightarrow e_2$ (event e_1 is causally ordered before event e_2). TFI first sets a hook to intercept event e_1. At run-time, as e_1 is intercepted, TFI blocks the execution of e_1, clears the hook for e_1, sets another hook to intercept e_2, and then resumes the execution of e_1. That procedure ensures that as long as event e_2 happens after event e_1, regardless of the number of intermediate events between e_1 and e_2, TFI will be able to intercept e_2, and then inject fault f at the moment e_2 is intercepted.

TFI can operate in distributed environments. Figure 3 illustrates the architecture of TFI, which includes three parts: a Local Controller (**LC**), a Central Controller (**CC**), and a global state BookKeeper (**BK**). Each LC attaches itself to a target component: (i) to intercept events of interest and (ii) to inject faults into that component. All LCs update the global BK with the local states

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems

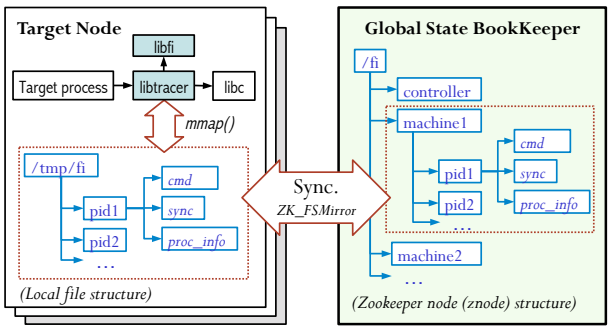


Fig. 4. Global and local state synchronization.

Fig. 5. Syntax and structure of TFI specification file.

of target components and receive commands from the CC (also through the BK). The CC, based on the global state maintained in the BK and instructions from a fault injection specification file (a fault injection experiment plan is automatically constructed out of the specification, and the CC uses the experiment plan for its input), orchestrates a fault injection via sending of two types of commands to LCs: events interception and fault injection.

TFI operates in an event-driven mode through a notification mechanism provided by the BK. That design was chosen to meet the timing requirements of the event coordination and fault triggering. Suppose that the CC is waiting for event e_i to occur in a component A, and it requests the BK to deliver a notification when the state of A is updated. When e_i is intercepted in A, the corresponding LC updates A's state in the BK. As a result, the BK immediately notifies the CC about the occurrence of e_i. Upon receiving that notification, the CC decides on the next command to be sent according to the fault injection experiment plan.

3.3.2 Implementation of TFI

The main challenges in implementing TFI are (i) minimizing side-effects of event interception and global state synchronization, and (ii) supporting large-scale distributed systems. We implemented the LC as a light-weight wrapper around the LibC library to intercept system-level events (e.g., Libc calls). Also, the LC updates local state via memory mapping (i.e., using *mmap*() function in Linux) to local file systems, an that action is automatically mirrored to the global state BookKeeper, backed by a ZooKeeper server [20] for scalability.

BookKeeper: State Synchronization via ZooKeeper. Figure 4 shows how global states are organized in ZooKeeper. ZooKeeper [20] is a scalable and highly available coordination system for distributed applications. It provides an abstraction of a hierarchical name space, like a virtual file system, to access its data. In our data model, each target system consists of multiple target machines; each target machine (virtual or physical) consists of one or multiple target processes, represented as their process IDs, `pid`. Each process has (i) a `cmd` znode (data entry in Zookeeper) to carry the current command that the CC sends to that process; (ii) a `sync` znode for synchronization back and forth between the LC attached to that process and the CC (e.g., the LC updates the content of the `sync` znode to notify the CC that the current command has been completed); and (iii) a `proc_info` znode to carry the information about the process.

ZooKeeper also provides a notification mechanism that clients can use to watch for updates or accesses in a Zookeeper znode. That mechanism is convenient for implementing the event-driven operation of TFI. When the CC sends a command by updating the *cmd* znode of a target process, it registers to watch the *sync* znode of that process. Once the LC attached to that process completes executing the command, it updates its *sync* znode so that the ZooKeeper server can send a notification to the CC. The CC's event handler is responsible for executing the subsequent commands in the experiment plan.

Local Controller: Light-weight Libraries Attached to Target Processes. The LC is implemented as two libraries, *libtracer* and *libfi* (as shown in Figure 4), which are attached to target processes. *libtracer* is a wrapper of the LibC library for intercepting Libc calls invoked by target processes. *libfi* implements fault models, e.g., process crash, process deadlock, and message corruption. *libfi* can be extended to include new fault models.

In order to reduce side-effects to target processes, we avoid using intensive network and file I/O operations within *libtracer*. *libtracer* writes local states to the local file system via memory-mapped (i.e., using *mmap*()) operations. Updated files are monitored by the *ZK_FSMirror* daemon, which instantly mirrors accesses back and forth between a specified local directory and a structure of ZooKeeper znodes. We utilize Linux's inotify [21] to monitor of file system events.

Central Controller: TFI Orchestrator. The main responsibilities of the CC are orchestration of causal event tracing and fault injection at predetermined locations. The CC also keeps track of the target machines and target nodes' liveness through monitoring of updates to ZooKeeper znodes. The CC interprets fault injection plans given in a fault injection specification file to automate the FPDB construction. The structure and syntax of a specification file are described in the next section.

3.3.3 TFI Specification Language

We define a specification language to facilitate the automation of TFI experiments, and make it easy for users to understand and interpret the context of a fault location returned by the FPDB.

The process of setting up a fault injection essentially consists of answering the following three questions: what, when, and where should the faults be injected?". The *what* asks for the fault model (e.g., process crash or message corruption) to inject. The *when* asks for the location in the program execution (e.g., after a program state changes to a particular value) at which the fault should be injected. The *where* asks for the target (e.g., the scheduler of the target system).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems

The specification language has been designed to mimic the TFI setup process. We defined three keywords, `inject`, `at`, and `to`, that users can use to provide answers to the three *what*, *when*, and *where* questions, respectively. The `inject` field is used to specify a fault model. Currently, our TFI supports three fault models, namely `ProcCrash`, `ProcDeadlock`, and `StringCorruption`. The `at` field is a sequence of causally ordered events. Each event is of one of the predefined event types: `MatchFunctionArg`, `MatchSyscallInvocations`, `MatchProcCount`, `Timer`, `VMStatus`, `HostStatus`, and `NetworkStatus`. For example, a `MatchFunctionArg` type of event is triggered when the specified arguments of a function matches a given value or pattern. The `to` field is used to specify a target, e.g., a process, for the fault. The specification uses the JSON (JavaScript Object Notation) format.

Figure 5 shows an example of a fault injection experiment specification file. At the highest level, there is an `experiment` object, which contains two injections. In the figure, the `NetworkingNetw` injection is highlighted. That injection can be interpreted as follows: `at` the end of the sequence `OsDoneBooting` → `NovaBootBegin` → `TaskNetworking`, `inject` a `ProcessCrash to` the `Scheduler` process. The `OsDoneBooting` event (type `MatchProcCount`) triggers when all of OpenStack's eight processes have been launched. The `NovaBootBegin` event (type `MatchFunctionArg`) is triggered when the `nova-napi` process receives an HTTP POST message (i.e., matching the "`POST*`" pattern). Finally, the `TaskNetworking` event is triggered when the `conductor` process writes to the database the message matching the following pattern: "`*UPDATE* task_state='networking*'`".

4 Limitations and Discussion

(1) Failures that we cannot diagnose. In general, our technique is not effective for failures that do not alter the processing flow of a request as defined in the context of this work. For example, a bug that only causes performance degradation, e.g., a slowdown in some part of a processing flow, cannot be successfully diagnosed by our technique. In order to diagnose performance-related issues, we need to develop a method to incorporate timing information into the current processing flow format. Our method also suffers from difficulties in finding faults that occur toward the end of a processing flow, because for such faults, most of the faulty processing flow is similar to normal execution. In order to resolve that issue, we would need to use finer-grained events, e.g., at program function call or basic block granularity. Also, if the fault type in production is not one of the types we used in the fault injection, our method will not be able to provide accurate root cause of the problem. To deal with this, new fault type can be easily incorporated into the fault injection and FPDB construction.

(2) Large number of entries in the database. Various factors affect the size (i.e., the number of failure profile entries) of the FPDB. Examples include the various types of requests, the length of the processing flow for each type of request, and the system configuration. It might be infeasible to construct an FPDB that covers all scenarios of a large-scale software. But we envision that each FPDB is created for a specific production system, with a stable configuration, to help developers quickly diagnose that particular deployment.

(3) Our diagnosis method does not eliminate the need for manual investigation. This is a common limitation of all fault localization techniques: they require developers to select the right answer based on the ranking (in our case, the trace distance) provided by the localization method. We further acknowledge that the locations we present to developers are only indicators of a context, in which a bug might occur. Similarly, an error message plays a role as an indicator of the context of a bug. As evaluated, our method provides a much better bug indicator than OpenStack's current error reporting mechanism. Furthermore, as pointed out by human studies [14], [33] on the effectiveness of fault localization techniques for bug fixing, developers not only need to know the location of a bug, they need to *understand* the surrounding context that triggers the bug. The more complex the system is, the more effort it takes to understand the context [14]. Thus, a good bug context indicator is always useful to developers.

(4) Supporting multiple distributed systems. When applying this failure diagnosis technology to a number of systems, it is required to build the FPDB for each distributed system. This is not a big burden in practice because each deployed system with its current deployment configuration is carefully maintained by its administrators in the real world, and an important while difficult and time-consuming task of the maintenance is the failure diagnosis. Our technology provides great benefits to failure diagnosis, and largely facilitates the administrators' current job. The administrators can decide the granularity of the injected faults when applying this technology; a small number of entries in FPDB, as a result of coarser granularity of fault injection, may also handle excessive number of failures though the granularity for helping localize the root cause place is coarser.

(5) Similar traces imply similar faults? Section 5.4.2 studies our main hypothesis "similar traces imply similar faults". The results show that more than 80% of injected similar faults lead to execution traces with their differences, or variations, less than 4%. Therefore, the false positives of our failure diagnosis, indicated as the real root cause is not close to the top 5 returned causes, is low (lower than 20% in Section 5.5). The high probability of the hypothesis holding true is because, though concurrency in distributed applications and their underlying systems tend to bring non-determinism and different execution traces, many non-determinisms are actually irrelevant to the application logic and can be pruned (as shown in Section 3.2.1), and a majority of faults actually do not interfere with the concurrency of the application logic itself since most parts of a distributed application are still its individual components' sequential executions. Of course, if the fault type is closely related to the concurrency of the application logic, e.g. the deadlock fault, the probability of this hypothesis holding true is reduced (but still quite high - more than 70% of deadlock faults lead to traces with variations less than 4% in Figure 9, because usually there are only a small number of concurrency modes across the fault-related components, i.e. only a small number of deterministic execution traces).

5 Evaluation

5.1 Settings and Goals

For evaluation, we use the OpenStack [31], a popular IaaS (Infrastructure-as-a-Service) cloud platform. Openstack is suitable for our evaluation since it is highly distributed and incorporates several software paradigms in its design. One instance of standard deployment involves at least 15 or more distributed components, and it utilizes multi-threaded architecture, event-based architecture and asynchronous message queues. Furthermore, a new version of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems

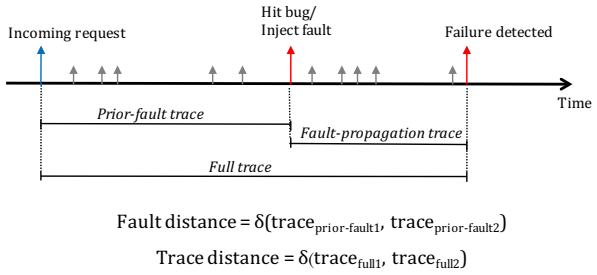


Fig. 6. Illustration of the fault distance and trace distance metrics. δ is the function to compute trace distances (Section 3.2.4).

OpenStack is released every six months. Each release introduces numerous new features and bug fixes.

These are the goals of our evaluation. First, we report on how effective current Openstack's error logging system is under crash failures in Section 5.4.1. Then, we verify the validity of the main hypothesis of our approach: *similar execution traces imply similar faults* (Section 5.4.2). Next, we study the accuracy of failure diagnosis against FI-induced failures (Section 5.5). Finally, we demonstrate how the FPDB helps localize the real-world bugs (Section 5.6).

From the above description of the technologies, including the flow construction for distributed applications, failure profile representation and profile comparison, and the target fault injection, one can see that these technologies are all generally applicable to distributed applications, and none of application specifics are associated with the technologies. In this paper we choose to give an in-depth evaluation of our solution in multiple aspects for the OpenStack application, though the solution can be generally applied to other distributed applications.

5.2 Evaluation Metrics

We introduce the concept of *fault distance* and *trace distance* in measuring the accuracy of diagnosis results. The *fault distance* is defined as the distance between two traces' segments starting from the event of incoming request and ending with the trigger event that triggers the fault injection. The *trace distance* is the distance between two given full traces. The distance is computed using the δ function explained in Section 3.2.4. These concepts are illustrated in Figure 6.

In measuring the *fault distance*, we discard all the events that follow after the triggering of the fault, leaving only the segment called *prior-fault trace*. For example, in a trace obtained from the fault injection, the *prior-fault trace* is the part of the trace containing events from the beginning of the full trace to the trigger event used to trigger the fault injection. In a trace from an actual fault, or bug, the fault trigger location is determined by the event closest to the first execution of the code region that contains the fault/bug (this information is derived from the actual code that is used to fix the bug). The granularity of a code region varies case-by-case. For example, a buggy code region can be a statement, a basic block, or an entire function. The description of the case study 2 in Section 5.6.3 exemplifies such a code region.

It is noteworthy to distinguish between the trace distance and the fault distance. The *trace distance* is used by the FPDB to infer the nearest faults (see Top-K query discussed in Section 2.3). In our experiment, once nearest faults are returned by the FPDB, we use *fault distances* to quantify their accuracy in terms of how close the returned faults are to the actual faults.

TABLE 1
Injection Locations for VM provision request (|**Faults**|: number of faults, |**Traces**|: number of collected traces)

Fault Type	Location Type	|**Faults**|	|**Traces**|
Process Crash	All monitored LibC calls	23,323	116,589
Message Corruption	All *read*, *write*, *send*, and *recv* LibC calls	18,221	91,092
Deadlock	All process- and lock-related LibC calls	2,143	10,702

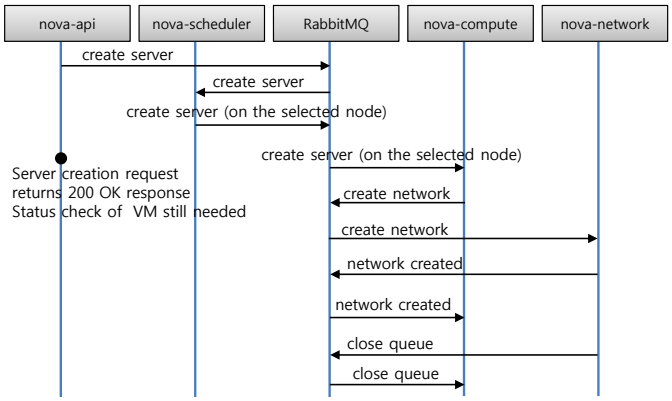


Fig. 7. A simplified message flow between OpenStack Nova components for a virtual machine provision request. Each arrow represents a pair of *send()* and *recv()* function calls in the source and destination components, respectively.

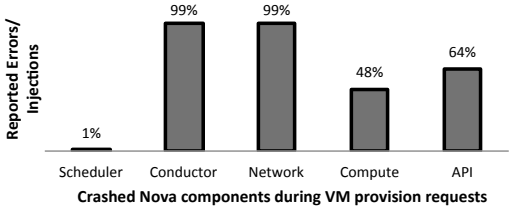
5.3 Construction of FPDB for OpenStack

We have constructed the FPDB for the Grizzly-1 version of Openstack, which was released in Jan 2013, following the procedure described in Section 2.2. Our FPDB contains fault injection traces for three types of requests in OpenStack, namely VM provision, VM resize, and VM migration. The selection of those requests was based on our analysis of the most frequently reported buggy requests for this OpenStack release.

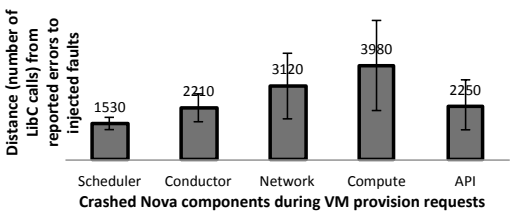
To construct the processing flow of each request, we have profiled eight of OpenStack's distributed components. They include five Nova (compute services) components, two Glance (storage service) components, and one Keystone (identity service) component. Figure 7 visualizes the profiled flow of a VM provision request within Nova's components. For further detailed analysis of the processing flows, we refer readers to our previous work [8].

Next, we conduct fault injection experiments to obtain the FPDB data. Specifically, we injected three types of faults in the experiments: process crash, message corruption, and deadlock. They are selected because they are typical examples of fault categories with different error propagation and manifestation characteristics. Process crashes usually have very short error propagation and have fail-silent behavior. Message corruptions propagate errors from one process/thread to other processes/threads and potentially cause other processes/threads to fail. Deadlock is also a type of fault with error propagating across processes/threads, but it has special failure manifestation, i.e. process/thread hangs. Process crash is injected by killing the process at the specified location and occasion; message corruption is injected by intercepting the message-involved LibC calls (read, write, send and recv) and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems



(a) The coverage of error messages.



(b) The lengths (number of LibC calls) of fault-propagation traces of failures that resulted in at least on error message in OpenStack log files.

Fig. 8. Evaluation of OpenStack's logging mechanism against process-crash injections.

flipping a bit in the message; a deadlock is injected by intercepting the process- and lock-related LibC calls and an infinite loop is executed during the interception.

We set the injection granularity to be at the level of LibC function calls, which resulted in 23,323 potential fault injection locations along the flow of a VM provision request. Table 1 summarizes how the locations are used to conduct fault injections for collecting failure profiles. For each fault, we collected five traces. We filtered out those cases where injections did not complete properly or the collected data were corrupted. Further investigation would be required to determine how glitches in our experimental setup caused the incompletion of those cases.

5.4 Analyze TFI Experiments

5.4.1 Resiliency of Openstack's Error Logging

OpenStack provides an embedded logging mechanism to aid developers in troubleshooting. Error messages in log files are often the first thing a developer or operator looks for when dealing with a failure. From the fault injections we conducted on Openstack we were able to learn how robust the Openstack's logging mechanisms were on crash failures. For this analysis, we used the results from the process crash fault injection (see the "Process Crash" row in Table 1). After the target process is killed, we collected all the logs from all components and searched for any error logs. Figure 8(a) shows the percentage of cases that have generated one or more of error logs to the number of all conducted injections for five nova components. The ratios are strikingly low for the nova-scheduler, nova-compute, and nova-api components: only 1%, 48%, and 64% of the crashes in these three components are recorded as errors in logs, respectively. Conversely, 99%, 52%, and 36% of the crashes in these components fail silently, respectively. This observation implies that developers might face great challenges while debugging a problem. They would need to delve into a large number of log files across many machines to figure out where to start diagnosing a failure.

For all failures represented by at least one error message in the log files, we calculated the length of the fault-propagation traces

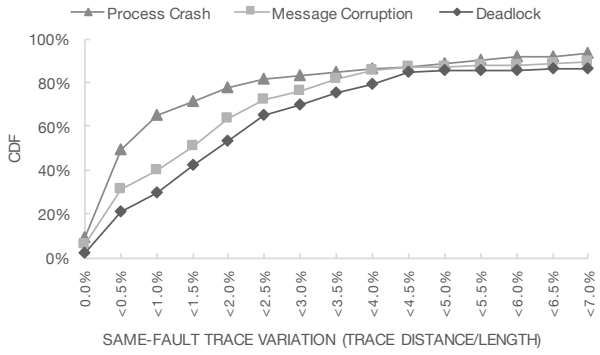


Fig. 9. Execution trace variations of failures caused by the same faults.

(see Figure 6, it is the number of libc function invocations from the fault injection point to the *write()* event of the first error message in the log files). Figure 8(b) shows that the average lengths are on the order of thousands of libc calls, ranging from 1530 to 3980. We learn that *nova-scheduler* does not generate an error log most of the time, but when it does, it reacts to the fault quicker than other components. In the following sections, we use these distances as a baseline to compare with the fault locations from the FPDB.

5.4.2 Variations of Execution Traces under Fault Injections

In this experiment, we have evaluated the accuracy of our method in terms of how close the predicted fault locations are to the actual fault location. We observed that repeated runs of the fault injections with the same fault (including both fault type and location) tend to generate similar failures. Specifically, there was less than 4% *variation* in their execution traces. We define the variation of two execution traces originated from the same fault as the ratio of their trace distance and the length of the shorter trace. Suppose we have two distinct traces t_i and t_j $(i \neq j)$ obtained from injecting the same fault, the trace variation is:

$$variation(t_i, t_j) = \frac{\delta(t_i, t_j)}{min(len(t_i), len(t_j))}$$

Figure 9 shows the largest variation between traces caused by the same injected faults. To obtain this measure for an injected fault, we computed the pair-wise variations of all five traces generated by that fault type, and then selected the largest one. The result shows that more than 80% of all the injected faults, across all three fault models, generate less than 4% of the trace variation. This implies that the similarity of execution flows are good indicators of the similarities between fault types across components. The process crash and deadlock faults generate the least and most trace variations, respectively. We attribute that to the more nondeterministic behavior of deadlock failures.

5.5 Accuracy of Failure Diagnosis against FI-induced Failures

In this section, we describe our analysis on the accuracy of our method in terms of the correct identification of the fault type, the affected component and how close the predicted fault locations are to the actual fault locations. We consider two scenarios: *known faults* and *unknown faults*. A *known fault* is a fault that has at least one trace generated by that fault in the FPDB. Conversely, a *unknown fault* does not have any trace generated by that fault stored in the FPDB.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems

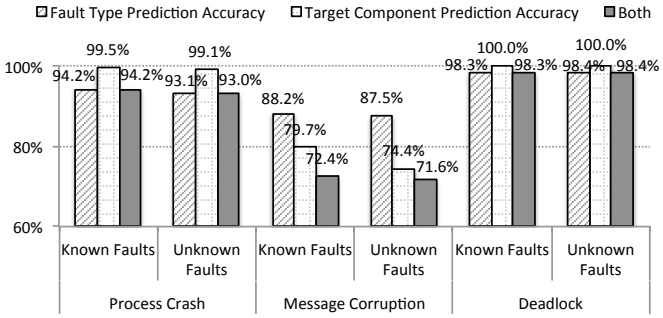


Fig. 10. The fault model and targeted component prediction accuracy of nearest-fault queries.

In order to evaluate FPDB queries against the *known fault* scenario, for each injected fault, we randomly removed one trace (out of five generated traces) from the original FPDB. The remaining FPDB contains four traces from each injected fault. The removed traces then were used to query against the remaining FPDB. For the *unknown fault* scenario, we randomly removed a half of all the injected faults (each holding traces of five repeated runs) stored in the original FPDB. The remaining FPDB contains no traces generated from the removed injected faults. For each removed injected fault, we chose one trace (out of five generated traces) to use as a query trace against the remaining FPDB. This set-up allows us to validate how accurate our FPDB-based diagnosis method can identify failure root-causes (e.g., the correct fault model, affected component, and the fault location) even when the FPDB does not contain traces of the exact fault of a query.

Figure 10 presents the accuracy of the nearest-fault queries (i.e. $K = 1$ in the Top-K query). It plots the accuracy results for the total of six cases drawn from *known/unknown* cases and three fault models – 'process crash', 'message corruption' and 'deadlock'. For each of the six cases, we show the accuracies in identifying the true fault model, true affected component, and both. In determining the true fault type and the affected component, we issue queries using the query sets for the *known* and *unknown* cases. The query returns from the FPDB the trace that is the closest to the query trace and we compare the fault type and the component of the returned trace to those of the query trace. According to our results, the accuracy lies in the range of 93-100% for the 'process crash' and 'deadlock' failures. However, the accuracy for the 'message corruption' failures achieved lower performance, 72.4% and 71.6%, for the *known* and *unknown* cases. That may reflect the longer fault-to-failure propagation paths caused by message corruption faults. Comparing the accuracy performance in terms of *known* and *unknown* cases, the accuracy of the *unknown* case tends to be slightly lower. However, this small difference between the *known* and *unknown* cases suggests that our TFI approach is robust in finding the root cause of the cases that had not been seen during the training. It is our future plan to conduct further detailed study on this, but this result suggests that the similarity of traces is high for the same type of faults that has the same root cause. Overall, with average accuracy rate of 88%, our TFI technique demonstrates the efficacy of identifying the fault type and the component.

For all the cases where the targeted components were correctly determined, we computed the fault distances which are the distances between the actual location of the injected fault and the fault location determined by TFI. If the fault location from TFI is close to the actual fault location, it would help developers quickly narrow down to the location along the request flows to look for the root cause of the fault. Figure 11 shows the results of the Top-K nearest fault queries with different K values (the number of returned faults nearest to the reference failure profile). For the *known* cases, about 50% of the Top-5 query results contained the exact fault locations.When K is 10 and 20, the accuracy increases to 74% and 77%, respectively. When we consider approximate fault locations (i.e., we do not require that exact fault locations be returned), about 67%, 81%, 82%, and 86% of the Top-1, Top-5, Top-10, and Top-20 queries, respectively, returned locations that are within 20 LibC calls from the actual faults. For the *unknown* cases, although the database could not pinpoint the exact fault locations, the accuracy of the provided fault locations was only slightly lower than for the *known* cases. That accuracy is two orders of magnitude better than that of OpenStack's existing error message mechanism, as shown in Figure 8(b).

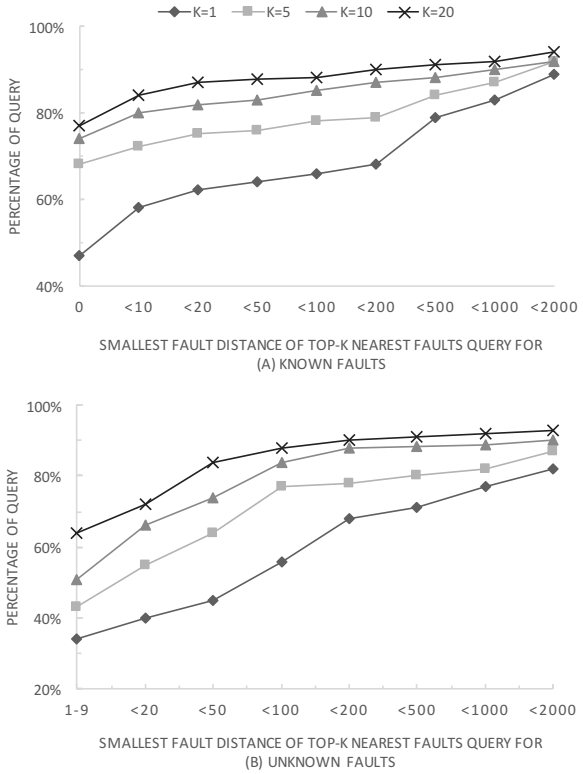


Fig. 11. Fault distance results of Top-K nearest fault queries with (a) known and (b) unknown faults.

5.6 Accuracy of Failure Diagnosis against Real Bugs

5.6.1 Methodology

We evaluate the FPDB-based failure diagnosis method against real bugs reported by OpenStack's users. We have collected several real bugs from the list of bug fixes published in the release note of the OpenStack Grizzly 2013.1 version (the major OpenStack release right after Grizzly-1 – the version that was used to construct the FPDB in Section 5.3). Because we could not conduct an experiment in a production system with the tracing tool enabled, we had to manually reproduce each bug and then capture its traces in our testing environment. In order to reproduce each bug, we have set up an experimental Openstack environment running the Grizzly-1 release, and installed the flow tracing tool sets.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems

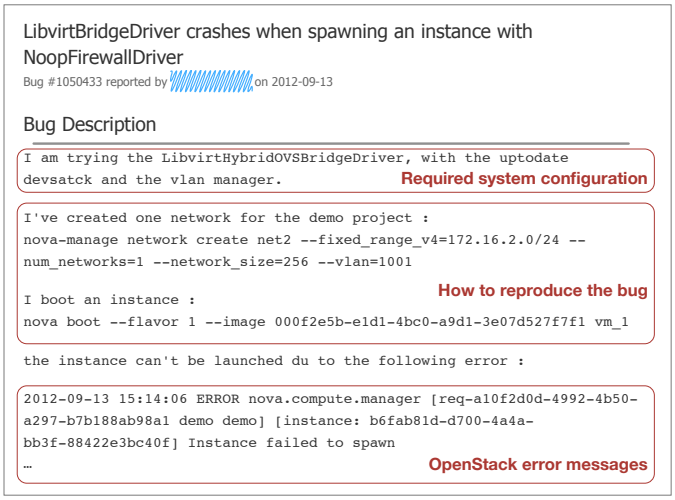


Fig. 12. An example of a typical bug report of OpenStack. A report usually provides information on how to reproduce the bug (e.g., system configuration and a sequence of commands) and a description of the system's behavior when the bug was observed (e.g., error messages). The first case described in Section 5.6.3 shows how this bug can be diagnosed using the FPDB approach.

For each bug, we relied on users' bug reports[2] to figure out how to reproduce the bug. Figure 12 shows an example of a typical bug report we studied. A typical bug report follows certain formats that has descriptions about what the required system configurations are, how to reproduce the bug and what the error message looks like. To determine the location of the bug, we manually inspected the developer's patching code for the bug. We used those bug locations as the ground truth to compare against the locations returned from querying our FPDB.

We chose bugs for the case study based on the following criteria. First, the bug had to be in the category of *high* or *critical* importance (as rated by OpenStack developers). Out of 766 bug fixes in total, we selected 208 bugs in that category, and ignored less critical bugs. Second, the requests to reproduce the bug had to be one of the three request types (i.e., VM provision, VM deletion, and VM migration) that we used to construct the FPDB. And finally, the reported OpenStack configuration to reproduce the bug had to match the configuration (e.g., we used the KVM hypervisor and MySQL database) that we used to generate the FPDB. For example, a reported bug that required PostgreSQL database had to be filtered out. Overall, 14 bugs out of 766 matched our criteria (i.e., they had the potential to affect our target system) and were manually reproduced in our experimental system.

In order to quantify the effectiveness of the FPDB, we used the Top-10 query to find injections that generated the closest traces to each reproduced trace of the selected bugs. We then computed the *fault distances* (in terms of the number of LibC function invocations) between each returned injection location and the locations where the code was fixed for each bug. We obtained the locations of the fixed code in the execution flow of a request by: (i) manually inspecting the commits that fixed the bug and then (ii) manually annotating the OpenStack source code to mark (i.e., print out a special message) at the beginning of the bug fixed area during OpenStack execution.

2. When users of the OpenStack originally encountered failures, they reported them to developers through the bug reporting system at https://bugs.launchpad.net.

5.6.2 Query Results

Table 2 summarizes the fault distance between the actual location and the location determined by our FPDB. In total, we have reproduced six VM provision bugs, four VM resize bugs, and four VM migration bugs. Eight out of 14 queries returned at least one trace that had fault distance within 100 LibC invocations. More importantly, the returned fault-propagation paths provided useful indicators for locating and fixing the actual bugs. The next section describes four cases in detail as to how our technique helps find the root cause of the bugs.

5.6.3 Examples of Localizing Real Bugs

This section describes the manual root-cause identification process of four reported bugs based on the results from querying the FPDB. For each bug, the manual process starts from inspecting the FPDB querying results. We used Top-10 query, each result consists of 10 injected fault locations, described in the TFI specification language. We then looked closer at the fault types and the OpenStack source code corresponding to these injected locations. If the returned injected fault locations are close to the location of the actual bug, the source code inspection tends to quickly lead to the discovery of the root-cause of the actual bug as we describe below.

Since most OpenStack components are written in Python programming language and executed directed on Python runtime interpreter, we customized our `libfi` to print out the stack call at fault injection points. Recall that `libfi` is a dynamic library attached to target processes at runtime to inject faults (see Fig 3 and Fig 4). The output stack calls, also stored in the FPDB, help us quickly map the injected fault locations to the affected OpenStack source code.

- **Case 1:** This bug is about `LibvirtBridgeDriver` crashing when trying to spawning a VM instance with `NoopFirewallDriver`[3]. Figure 12 shows the main content of the actual bug report. The following error message is printed if a user attempts to issue the request.

```
libvirtError: Network filter not found: Could not
    find filter 'nova-instance-instance-xxx-xxx'
Instance failed to spawn.
```

How our technique was useful: In the FPDB, the closest failure profile to that bug was generated by injecting a message corruption fault to one of the `write()` LibC function call invoked by a `nova-compute` process. This `write()` call is the one used by the `nova-compute` to send a network filter configuration to the socket of `libvirtd`, the component that directly executs the VM provision task. During our targeted fault injection, the name of the network filter was corrupted as shown below:

```
Original value:
nova-compute 7fad40b08700  6208    write (15, AF_FILE
    =>/var/run/libvirt/libvirt-sock) = 74: <filter
    name='nova-instance-instance-ID-MACADDRESS'
    chain='root'>

New value (the 33rd character is corrupted):
nova-compute 7fad40b08700  6208    write (15, AF_FILE
    =>/var/run/libvirt/libvirt-sock) = 74: <filter
    name='nova-instance-instbnce-ID-MACADDRESS'
    chain='root'>
```

3. Bug report: https://bugs.launchpad.net/bugs/1050433

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2575829, IEEE Transactions on Parallel and Distributed Systems

TABLE 2
Evaluation results with real OpenStack bugs: Smallest fault distances of top-10 queries

Request type	VM Provision						VM Resize				VM Migration			
Smallest fault distance (in LibC calls)	**2**	14	**18**	221	236	565	**14**	56	84	245	**34**	45	342	442
Described case number (Case #)	**1**		**4**				**2**				**3**			

By tracing back the data flow of that message content in the execution of `nova-compute`, we were able to immediately discover that the function that generated the VM network configuration was directly involved in constructing the message.

It turned out that this function generated a configuration for a network filter regardless of whether the `NoopFirewallDriver` option was selected or not. However, the `NoopFirewall` driver was not implemented to recognize such network filter configuration. Thus it raised an exception and aborted the VM provision request. The solution was that the network configuration function needed to be fixed to validate the `NoopFirewallDriver` option before generating a network filter configuration.

• **Case 2:** This case is about the VM deletion failure when VM instance is in `RESIZED` state[4]. No error messages are generated.

How our technique was useful: In the result of the FPDB query against the reproduced trace of this bug, there were two faults indicating that the `nova-compute` process crashed during the delete action: one fault before and one fault after the VM status was checked for `RESIZED`. Based on these hints, we focused our inspection to the code that verified the `RESIZED` VM status. Our inspection confirmed that this is indeed the offending code that was causing the failure.

```
...
if instance['vm_state'] == vm_states.RESIZED:
  # If in the middle of a resize, use confirm_resize
  # to ensure the original instance is cleaned up
  # too
  migration_ref =
    self.db.migration_get_by_instance_and_status(
      context, instance['uuid'], 'finished')
...
```

Specifically, if the check for the RESIZED returned true, the delete API had to wait for the VM to reach a safe state, e.g., to finish resizing (because a VM could not be deleted while it was resizing). While waiting, function `migration_get_by_instance_and_status` periodically made queries to the MySQL database to determine the VM's state. However, those database queries required higher context privileges (given by the `context` parameter) than the privilege at which the delete API was executing. Thus, the query failed. This bug can be fixed by elevating the context privilege of the current call (e.g., changing the `context` parameter to `context.elevated()`).

• **Case 3**: This bug is the case in which the VM migration fails when multiple requests are issued at the same time[5]. When three migration requests are issued concurrently, one of them might fail, and OpenStack periodically prints the following error message.

```
[instance: xxx] 'old_instance_type_memory_mb'.
    Setting instance vm_state to ERROR
```

How our technique was useful: Our FPDB query showed that the reproduced trace of this bug was similar to several traces generated by the faults injected during the *VM resize* request. We focused our inspection to the code area that was close to these fault-injected locations returned from the FPDB. In this case the code region was the logic that handled the VM migration. We learned through the inspection that majority of the codes that handled the VM migration request was being shared with that of the VM resize request. Typically, a check is used to differentiate between those two request types when needed. In this reported bug, the request type check was omitted, thus leading to the wrong execution of the VM resize code path, which was determined correctly by our query.

• **Case 4**: This bug is caused by an invalid `availability_zone` parameter for the nova boot. If an invalid `availability_zone` is specified, instead of printing a normal error message to notify users that the parameter is invalid, OpenStack continues to process the request until it could not find a valid host machine for the VM. Finally, OpenStack would print a NoValidHost error message to notify the user that it could not provision the VM. However, this error message is not very helpful because this error message is produced from many other root causes as well.

How our technique was useful: According to our FPDB query results, there was one failure profile similar to this bug generated by injecting a message corruption fault to a `send()` LibC function invocation in the `nova-schedule` process. We looked at the code of this `send()` and learned the followings. This `send()` call is used to send a request to MySQL to query a list of available hosts based on the user input filter, which includes the `availability_zone` parameter. If one of the filter criteria is modified to an invalid value, the database returns an empty list, which is the same behavior exhibited by the bug.

Our FPDB result directly led us to the code that needed to be fixed. All parameters should have been sanitized to disallow invalid parameters before the execution had sent a message to the database that would return the same value as a corrupted message. We examined the input sanitization code and found that the `availability_zone` parameter was not correctly validated.

6 Related Work

Distributed Tracing: Aguilera et al. proposed methods for inferring the causality of messages using logs and demonstrated how the method can be used for performance debugging [2]. Anandkumar et al. model the message traces using semi-Markov process and match them probabilistically [4]. However, for targeted fault injection, we require knowledge of precise traces so that we can plan the fault injection experiments. Further, we can classify techniques that provide precise tracing results, but requires instrumentation. Examples of instrumentation-based techniques are Magpie [6], Google Dapper [35] and X-trace [13]. We have employed the principles of vPath [38] in our targeted fault injection technique, it is one example of technique that does not require any application instrumentation.

Fault Injection: Fault injections are widely applied to evaluate resilience of distributed systems (including cloud platforms), and

4. Bug report: https://bugs.launchpad.net/nova/+bug/1056601
5. Bug report: https://bugs.launchpad.net/nova/+bug/1160489

a number of fault injection tools have been developed for that purpose. For example, NFTAPE [36] enables specification of fault injection, conducts injection experiments, and collects experimental results. Chaos Monkey [10] randomly injects faults into VM instances of cloud platforms. Failure-as-a-service [16] performs large-scale online failure injections in actual deployments of cloud services, and provides the fault injection as a cloud service. Those tools either conduct random fault injections, or perform fault injections at certain breakpoints determined by local-state information. Thus, they are not suited to our purpose.

There are also tools that inject faults in distributed systems based on global-state information. The authors of [9] present a global-state-triggered fault injector, and the authors of [18] designed a language-driven tool for injecting faults into distributed systems. In [9], the handling of global state, which bears high-level application semantics, is instrumented in target distributed systems for fault injection (an in-depth understanding of target systems is required). In [18], conditions associated with message flows or low-level stack contexts cannot be specified in the language. We aim to provide a general methodology and apply fault injection with targeted scenarios specified at a low-level message flow granularity. The above existing fault injection tools based on global states do not apply.

In our case study, we injected faults into OpenStack. A recent paper, [22], also discusses fault injection into OpenStack. However, the objective of that paper is to study the system's resilience, and high-level operations like REST (Representational State Transfer) APIs are instrumented manually to reconstruct the system execution graph so that faults are injected at certain points.

Failure Diagnosis Techniques: A large body of research focused on *fault localization* techniques [1], [34], [39], [40]. These techniques aim at determining the likelihood of each executable program unit (e.g., a program statement) containing a bug. Two main approaches have been proposed: program slicing [39] and spectrum-based [1]. Many variations of them have been developed to improve practicality. However, because of the intrusiveness and large amount of output data, those techniques are not amenable to direct application to large programs [32]. For a more comprehensive review of fault localization techniques, we refer readers to [40], which surveys the state-of-the-art research in this area.

The other approaches to failure diagnosis can be categorized as *failure grouping* [12], [15], [25], [28]. The common principle of these techniques is to classify failures into groups of similar symptoms, assuming that a common root cause is likely to generate similar symptoms across runs. Failure grouping reduces cost of diagnosis by either identifying recurrent or known failures [25], or directing resources to resolve higher impact failures, e.g., failures reported by a larger number of users [12], [15], [28]. However, these techniques cannot diagnose *unknown* failures. We combine the idea of failure grouping with fault injection to deal with unknown failures.

7 Conclusion

We have proposed and developed a method to assist failure diagnosis in distributed systems. The method is a novel use of fault injection to populate a database of processing flows of a target system executing under failures. The database is then used to help identify root causes of failures observed in the field by providing injected faults that generated similar processing flows. We showed how the method can help identifying real bugs in OpenStack.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proc. of the Academic and Industrial Conf. on Testing: Practice and Research Techniques - MUTATION*, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of the 19th Symp. on Operating Systems Principles (SOSP '03)*, pages 74–89, New York, NY, USA, 2003. ACM.

[3] AMQP – Advanced Message Queuing. http://amqp.org.

[4] A. Anandkumar, C. Bisdikian, and D. Agrawal. Tracking in a spaghetti bowl: Monitoring transactions using footprints. In *Proc. of the 2008 ACM SIGMETRICS Intl. Conf. on Measurement and modeling of Computer Systems (SIGMETRICS '08)*, pages 133–144, New York, NY, USA, 2008.

[5] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless seth is false). In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, STOC '15, pages 51–58, New York, NY, USA, 2015. ACM.

[6] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. of the 6th Symp. on Operating Systems Design & Implementation (OSDI'04)*, Berkeley, CA, USA, 2004. USENIX Association.

[7] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, pages 85–90, 2003.

[8] S. A. Baset, C. Tang, B. C. Tak, and L. Wang. Dissecting open source cloud evolution: An openstack case study. In *The 5th USENIX Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, 2013. USENIX.

[9] R. Chandra, R. M. Lefever, K. R. Joshi, M. Cukier, and W. H. Sanders. A global-state-triggered fault injector for distributed system evaluation. *IEEE Trans. on Parallel and Distributed Systems*, 15(7):593–605, 2004.

[10] Chaos Monkey. http://techblog.netflix.com/2011/07/netflix-simian-army.htm.

[11] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *ECOOP 2005-Object-Oriented Programming*, pages 528–550. Springer, 2005.

[12] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proc. of the 2012 Intl. Conf. on Software Engineering*, pages 1084–1093, Piscataway, NJ, USA, 2012. IEEE Press.

[13] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proc. of the 4th USENIX Conf. on Networked Systems Design & Implementation (NSDI'07)*, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[14] Z. P. Fry and W. Weimer. A human study of fault localization accuracy. In *Proc. of the 2010 IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 1–10. IEEE, 2010.

[15] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proc. of the ACM SIGOPS 22nd Symp. on Operating Systems Principles*, pages 103–116, New York, NY, USA, 2009. ACM.

[16] H. S. Gunawi, T. Do, J. M. Hellerstein, I. Stoica, D. Borthakur, and J. Robbins. Failure as a service (FaaS): A cloud service for large-scale, online failure drills. Technical report, EECS Department, University of California, Berkeley, 2011.

[17] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[18] W. Hoarau and S. Tixeuil. A language-driven tool for fault injection in distributed systems. In *Proc. of the 6th IEEE/ACM Intl. Workshop on Grid Computing*, pages 194–201. IEEE Computer Society, 2005.

[19] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. of the 2010 USENIX Annual Technical Conf.*, Berkeley, CA, USA, 2010.

[21] Inotify – Linux man page. http://linux.die.net/man/7/inotify.

[22] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva. On fault resilience of OpenStack. In *Proc. of the 4th Annual Symp. on Cloud Computing*. ACM, 2013.

[23] S. Kim, T. Zimmermann, K. Pan, and E. Whitehead. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2006*, pages 81–90, Sept 2006.

[24] G. Kroah-Hartman. Modifying a Dynamic Library Without Changing the Source Code. http://www.linuxjournal.com/article/7795.

[25] I. Lee and R. Iyer. Diagnosing rediscovered software problems using symptoms. *IEEE Trans. on Software Engineering*, 26(2):113–127, 2000.

[26] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, New York, NY, USA, 2006. ACM.

[27] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.

[28] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *Proc. of the 14th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering*, pages 46–56. ACM, 2006.

[29] C. Liu, X. Yan, H. Yu, J. Han, and S. Y. Philip. Mining behavior graphs for "backtrace" of noncrashing bugs. In *SDM*, pages 286–297. SIAM, 2005.

[30] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18 – 31, 1980.

[31] OpenStack. http://www.openstack.org/.

[32] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proc. of the 2011 Intl. Symp. on Software Testing and Analysis*, pages 199–209. ACM, 2011.

[33] D. R. Raymond. Reading source code. In *Proc. of the 1991 Conf. of the Centre for Advanced Studies on Collaborative Research*, pages 3–16. IBM Press, 1991.

[34] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. In *Proc. of the 8th Intl Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 139–152, New York, NY, USA, 2013. ACM.

[35] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[36] D. Stott, P. Jones, M. Hamman, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: Networked fault tolerance and performance evaluator. In *Proceedings. Intl. Conf. on Dependable Systems and Networks (DSN'02)*, page 542. IEEE, 2002.

[37] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *21st International Symposium on Fault-Tolerant Computing (FTCS)*, pages 2–9, June 1991.

[38] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. vPath: Precise discovery of request processing paths from black-box observations of thread and network activities. In *Proc. of the USENIX Annual Technical Conf.*, Berkeley, CA, USA, 2009.

[39] M. Weiser. Program slicing. In *Proc. of the 5th Intl Conf. on Software Engineering*, pages 439–449. IEEE Press, 1981.

[40] W. E. Wong and V. Debroy. A survey of software fault localization. *University of Texas at Dallas, Tech. Rep. UTDCS-45-09*, 2009.

Cuong Pham is a PhD candidate at UIUC. His research interests are in system reliability and security, virtualization, operating systems and cloud computing. He is the receiver of the William Carter Award in 2014 to recognize the important contribution of his graduate dissertation research to the field of dependable computing. Mr. Pham received his MS degree in computer engineering in 2013 from UIUC, and his BS from Hanoi University of Technology, Vietnam in 2007.

Long Wang is a Research Staff Member at the IBM T.J. Watson Research Center, Yorktown Heights, NY, where he leads the architecture of Disaster Recovery of IBM Cloud Managed Services to IBM Resiliency Services. His research interests include Fault-Tolerance and Reliability of Systems and Applications, Dependable and Secure Systems, as well as Measurement and Assessment. He obtained his Ph.D. degree from Department of Electrical & Computer Engineering in University of Illinois at Urbana-Champaign (UIUC) in 2010. Dr. Wang is a member of the IEEE.

Byung Chul Tak is a Research Staff Member at IBM T.J. Watson Research Center, Yorktown Heights, NY. He received his Ph.D. in computer science in 2012 from Pennsylvania State University. He received his MS degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST) in 2003, and his BS from Yonsei University, Korea in 2000. He is the recipient of IBM PhD Fellowship, an IBM Outstanding Technical Achievement Award and an IBM Research Division Award. His research interests include distributed systems, virtualization, operating systems and cloud computing. He is a member of IEEE.

Salman Baset is a research staff member at IBM T.J. Watson Research Center, Yorktown Height, NY since December 2010. He received a B.S. degree in Computer System Engineer from GIK Institute of Engineering Sciences and Technology, Pakistan in 2001, his M.S. and Ph.D. in Computer Science from Columbia University. He led the design of SPEC Cloud IaaS 2016, the first industry standard benchmark for measuring cloud performance. His research interests include containers, configuration, and security, as well as next generation analytics for IPTV and Telcos. He is a recipient of SPEC Presidential Award, Marconi Young Scholar Award, and several IBM awards. He is a member of IEEE and ACM.

Chunqiang Tang is a software engineer at Facebook. Dr. Chunqiang Tang received the B.E. degree from the University of Science and Technology of China, Hefei, China, in 1996, the M.S. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 1999, and the Ph.D. degree from the University of Rochester in 2004. He joined the IBM T.J. Watson Research Center as a Research Staff Member in August, 2004. In 2013, he joined Facebook. His research interests lie in the areas of information retrieval, distributed systems, computer networks, and operating systems. He is a member of IEEE and ACM.

Zbigniew Kalbarczyk is a Research Professor in the Coordinated Science Laboratory of UIUC. Dr. Kalbarczyks research interests are in the area of design and validation of reliable and secure computing systems. His research also involves design of techniques for automated validation and benchmarking of dependable computing systems using formal and experimental methods. He has served as a Program Chair of the International Conference on Dependable Systems and Networks (DSN), 2007 and as a Program Co-Chair of DSN 2002. Dr. Kalbarczyk has published over 90 technical papers. He holds a Ph.D. degree in computer science from the Technical University of Sofia, Bulgaria. He is a member of the IEEE, the IEEE Computer Society, and IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance.

Ravishankar K. Iyer is George and Ann Fisher Distinguished Professor of Engineering at UIUC. He holds appointments in the Department of Electrical and Computer Engineering and the Department of Computer Science. His research interests are in the area of dependable and secure systems. He has been responsible for major advances in the design and validation of dependable computing systems He has received several awards, including the Humboldt Foundation Senior Distinguished Scientist Award for excellence in research and teaching, the AIAA Information Systems Award and Medal for 'fundamental and pioneering contributions toward the design, evaluation, and validation of dependable aerospace computing systems,' and the IEEE Emanuel R. Piore Award for 'fundamental contributions to measurement, evaluation, and design of reliable computing systems.' He is a fellow of the AAAS, the ACM, and the IEEE.