2019 IEEE 12th International Conference on Cloud Computing (CLOUD)

Testing for Bugs of Cloud-Based Applications Resulting from Spot Instance Revocations

Abdullah Alourani, Ajay D. Kshemkalyani, and Mark Grechanik
University of Illinois at Chicago
Chicago, IL 60607

***Abstract*—One of the major advantages of cloud spot instances in cloud computing is to allow stakeholders to economically deploy their applications at much lower costs than that of other types of cloud instances. In exchange, spot instances are often exposed to revocations (i.e., terminations) by cloud providers. With spot instances becoming pervasive, terminations have become a part of the normal behavior of cloud-based applications; thus, these applications may be left in an incorrect state leading to certain bugs. Unfortunately, these applications are not designed or tested to deal with this behavior in the cloud environment, and as a result, the advantages of cloud spot instances could be significantly minimized or even entirely negated. We propose a novel solution to automatically find these bugs and locate their causes in the source code. We evaluate our solution using 10 popular open-source applications. The results show that our solution not only finds more instances and different types of these bugs compared to the random approach, but it also locates the causes of these bugs to help developers to improve the design of the shutdown process for cloud-based applications.**

***Index Terms*—cloud computing; spot instances; shutdown bugs; application bugs; kernel modules; irregular terminations of cloud-based applications; spot instance revocations**

I. Introduction

With spot instances becoming pervasive, irregular terminations have become a part of the normal behavior of cloud-based applications. ***B**ugs of cloud-based **A**pplications resulting from **S**pot **I**nstance **R**evocations (BASIR)* result from errors in the implementation of the shutdown instructions of these applications that occur only during spot instance revocations. When these applications are being irregularly terminated, they might lose their states that lead to BASIR, such as data loss, inconsistent states, performance bottlenecks, hangs, crashes, deadlocks, locked resources, or these applications that cannot restart/terminate. Cloud-based applications that run in spot instances (also known as spot virtual machines (VMs)) are not designed or tested to deal with this behavior in the cloud environment. The shutdown sequence of a cloud-based application is often left untested because developers often assume that a cloud-based application is properly terminated as long as its processes are terminated. It is very difficult to find BASIR because a termination signal can be initiated at every execution state of a cloud-based application, leading to a significantly larger search space of application states. Unfortunately, the absence of testing the effect of spot instance revocations on cloud-based applications will likely lead to a large number of BASIR. As a result, the advantages of cloud spot instances could be significantly minimized or even entirely negated.

In general, terminations could be seen as *regular* when an application receives a termination signal in the context of predefined protocols, or *irregular* when an application receives a termination signal without using any context of predefined protocols. Hence, the revocations of spot instances often lead to irregular terminations of cloud-based applications. Note that an application can be irregularly terminated in two modes. We assume that the reason for executing an application is to run an algorithm that implements the requirements of this application to provide the required results. First, an application could be irregularly terminated during the execution of the application's algorithm. Second, an application could be irregularly terminated during the execution of the shutdown sequence of the application when the execution of the application's algorithm is completed. Moreover, irregular terminations do not affect stateless applications but often affect stateful applications relying on the results of ongoing calculation by applications under irregular terminations. These stateful applications might change to incorrect states when they are terminated before their shutdown sequences are entirely executed. In general, resources utilized by an application under irregular termination can be called ***R**esources **A**ffected by **T**ermination (RAT)*. When an application (A) encounters irregular terminations while interacting with another application (B), B is considered RAT because it might be left in an incorrect state until it identifies that A is already terminated.

We propose a novel solution to automatically find BASIR and locate their causes in the source code of cloud-based applications. We develop our solution for ***T**esting for **BASIR*** *(`T-BASIR`)* that uses kernel modules (KMs) [1] to find these bugs and generate traces of their causes in the source code. These bugs and traces can be analyzed by developers, who look for fixes of these bugs to reduce or even eliminate the number of these bugs when cloud-based applications encounter irregular terminations. Our paper makes the following noteworthy contributions:

- We address a new and challenging problem for cloud-based applications that results from irregular terminations due to spot instance revocations.
- To the best of our knowledge, `T-BASIR` is the first automated solution to find and fix bugs of cloud-based applications resulting from spot instance revocations.
- We evaluate `T-BASIR` using 10 popular open-source applications. Our results show that `T-BASIR` not only

2159-6190/19/$31.00 ©2019 IEEE
DOI 10.1109/CLOUD.2019.00050

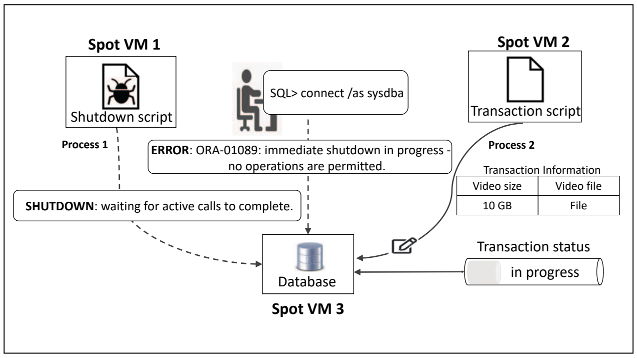

Fig. 1: An illustrative example of BASIR.

finds more instances and different types of BASIR (e.g., performance bottlenecks, data loss, locked resources, and applications that cannot restart) compared to the random approach, but it also locates the causes of BASIR to help developers to improve the design of the shutdown process for cloud-based applications.

- `T-BASIR`'s code and our experimental results are publicly available [2].

II. Problem Statement

In this section, we discuss sources of BASIR, illustrate the BASIR problem, and formulate the problem statement.

A. Sources of BASIR

There are two primary sources of BASIR. The first one is spot instance revocations. The revocations of spot instances are based on price fluctuations that happen based on demand of spot instances from many cloud customers. The cloud providers often revoke spot instances when the demand increases and the number of available spot instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases. It is very difficult to determine in advance spot instance revocations that depend on the varying demands of cloud customers. Doing so requires cloud customers (i.e., application's owners) to understand how the demands of the spot instances change, how the costs of the allocated spot instances change, and how to make trade-offs between the demands and these costs [3]. As a result, price fluctuations that depend on the demand have a high influence on the number of spot instance revocations.

The other source is shutdown bugs of applications. The shutdown bugs of applications often result from errors in the implementation of a cleanup process of these applications that occurs only during their shutdowns. It is very difficult to analyze irregular terminations, even for a single execution path of an application for certain inputs since termination signals can be initiated at every point during the execution of the path resulting in deviations from the execution path. For example, termination signals that are initiated during the execution of the third-party's instructions could change the application state, resulting in BASIR. Also, it is very difficult to specify in which sequence instructions should be executed during the shutdown of an application. Doing so requires the knowledge of the execution state of an application at any point when this application receives a termination signal. Furthermore, multiple termination signals can be initiated during the execution of the shutdown instructions of an application, leading to a significantly larger search space.

B. Illustrative Example

The BASIR problem with a cloud-based application is illustrated in Fig. 1. As discussed in Section II-A, BASIR results from two primary sources: shutdown bugs of applications and spot instance revocations. We show an instance of BASIR that arises from the interactions between a shutdown bug of an application, which comes from a real shutdown bug [4], and the revocation of a spot VM that represents the normal behavior of spot VMs. Our illustrative example shows a typical cloud-based application where a cloud-based application and its artifacts are often replicated across multiple VMs to improve its fault tolerance and reduce its network latency. The cloud-based application and its artifacts are deployed on three spot VMs, where spot VM 1 contains an Oracle shutdown script that reflects a routine script for databases in production, spot VM 2 contains a transaction script that uploads a video file with a large size (e.g., 10GB), and spot VM 3 contains an Oracle database.

Suppose that the Oracle shutdown script in spot VM 1 that runs on a particular process (Process 1) is executed to terminate the Oracle database that runs in spot VM 3 at the same time another process (Process 2) in spot VM 2 is holding the lock on this Oracle database to perform the transaction. Hence, Process 1 will be waiting until Process 2 releases the lock from the Oracle database. However, consider what happens when spot VM 2 is revoked as a part of the normal behavior of spot VMs while the transaction that is executed by Process 2 is still ongoing. Since Process 2 does not release the lock before the revocation of spot VM 2, the Oracle database will hang and consume needlessly resources until Process 1 determines that Process 2 is gone. The Oracle database prevents users from performing other operations (see the error message in the middle of Fig. 1), since the database is waiting for active calls to be finished (see the log on the left side of Fig. 1). Furthermore, if the spot VM 3 that contains the database is also revoked, this revocation (i.e., an irregular termination of the database) may not only produce an inconsistent state of various data or an incorrect state of artifacts in the database but also may affect the execution of subsequent instances of the database.

C. The Problem Statement

With spot instances becoming pervasive, bugs of cloud-based applications resulting from spot instance revocations have become a very important concern for cloud customers (i.e., application's owners). In this paper, we address a new and challenging problem of testing the effect of spot instance revocations on cloud-based applications – *how to find and fix bugs of cloud-based applications that result from spot instance* *revocations*. The root of this major problem is that cloud-based applications that are exposed to irregular terminations are not designed or tested to deal with this behavior in the cloud environment. Thus, when cloud-based applications are being irregularly terminated, their current state might be lost, which leads to certain bugs, such as data loss, inconsistent states, performance bottlenecks, hangs, crashes, deadlocks, or locked resources. On top of poor user experience from seeing these bugs, other bugs result in situations where cloud-based applications could not be restarted without manual interventions. As a result, the advantages of cloud spot instances could be significantly minimized or even entirely negated. To the best of our knowledge, there is no automated solution to find and fix bugs of cloud-based applications that result from irregular terminations due to spot instance revocations.

III. Solution

In this section, we introduce KMs, explain why we use KMs and describe how we utilize KMs in `T-BASIR`.

A. Why We Use Kernel Modules in `T-BASIR`

A KM is a mechanism for (un)loading some codes into an operating system at runtime without rebooting the operating system to extend its functionalities [1]. KMs facilitate modifying the flow of executions, handling the interruption of termination signals, and accessing the information of kernel space functions. There are three main reasons for using KMs rather than modules in the user space. First, using modules in the user space, it is very difficult to synchronize between a process of a cloud-based application that performs a specific operation (e.g., write) on certain resources and a process that sends a termination signal to this application. Second, it is very difficult to time the execution of a particular instruction of a cloud-based application in the user space because an operating system that runs in the kernel space determines the schedule of executing this instruction. Third, some termination signals (e.g., SIGKILL) often invoke the signal handlers in the kernel space instead of the signal handler in the user space (i.e., a signal handler that is defined in the source code of a cloud-based application). In contrast, KMs have complete control over the execution in the kernel space at runtime. As a result, `T-BASIR` uses KMs to ensure termination signals are sent to certain points in the execution of a cloud-based application and to measure the impact on the state of RAT at these points of the execution in order to find BASIR.

B. Automating BASIR Detection Using Kernel Modules

In `T-BASIR`, our terminator KM specifies when we send a termination signal during the execution of cloud-based applications that mimics the irregular terminations, as discussed in Section I. An essential goal is to identify which instructions of cloud-based applications are more likely to lead to BASIR in order to send termination signals during the executions of these instructions. Given that BASIRs are more likely to be exposed when instructions use resources to perform certain operations (e.g., write) that are often accessed when specific system calls (e.g., acquire-lock) are invoked, we favor instructions whose executions access these resources. Our terminator KM sends a termination signal during the execution of these system calls, which correspond to specific instructions in the source code. Our terminator KM uses the number of a system call with KProbe and JProbe interfaces [1] to intercept the execution of these system calls and, hence, ensures that a termination signal is sent to certain points of the execution. In summary, our terminator KM sends termination signals only during the execution of these instructions to increase the degree of precision for finding BASIR. In the RANDOM approach, a termination signal is sent to any point in the execution of a cloud-based application. Our hypothesis is that our terminator KM is more effective than randomly sending termination signals to any instructions because determining to which instruction a termination signal should be sent is highly correlated to the probability of finding BASIR. We verify our hypothesis with the experimental data in Section V.

In `T-BASIR`, our detector KM determines when irregular terminations lead to BASIR. We use the values of RAT (e.g., variables and artifacts) for cloud-based applications to identify the presence of BASIR. Initially, we randomly select a set of system calls of a cloud-based application. Then, we use our identifier KM to record the values of RAT that are used by these system calls when a cloud-based application is regularly terminated. For each system call, we run this application to collect the values of the RAT when this application is irregularly terminated. Our detector KM uses Eq. (2) to measure the difference between the value of RAT when the cloud-based application is regularly terminated and the value of the same RAT when the cloud-based application is irregularly terminated during the execution of the same system call. We use the difference operation to evaluate the presence of BASIR by analyzing executions between irregular and regular terminations, since we assume that running a single execution path of a cloud-based application for certain inputs multiple times leads to the same values of the RAT in different runs. When the value of the RAT after irregular terminations varies from the expected value of the RAT at the same point in the execution after regular terminations, it indicates a potential instance of BASIR. Hence, once a difference is found, the detector KM uses Eq. (1) to add this difference to the total number of potential BASIR and collects the traces of this BASIR, as discussed in Section III-C. As a result, developers can analyze the found instances of BASIR and their traces to improve the design of the shutdown process for applications.

$$B(T,T') = \sum_{i=1}^{n}\sum_{j=1}^{m} D(t_{ij},t'_{ij}) \text{ where } t \in T,\ \ t' \in T' \tag{1}$$

$$D\big(t_{ij},t'_{ij}\big) = \begin{cases} 0 & t_{ij} = t'_{ij} \\ 1 & t_{ij} \neq t'_{ij} \end{cases} \tag{2}$$

Here, T is a matrix of size $n \times m$, n and m designate the total number of system calls and RAT, respectively, for

Algorithm 1 `T-BASIR`'s algorithm for finding BASIR and locating their causes.

1: **`Inputs:`** KM Configuration Ω, Application $\mathcal{A}$
2: **`LoadIdentifierKMs`**(Ω)
3: **while** $\mathcal{A}$ $\neg$ **`Terminate`** **do**
4:     $\mathcal{T} \leftarrow$ **`IdentifySyscallRAT`**($\mathcal{A}$, Ω)
5: **end while**
6: **`UnloadIdentifierKMs`**(Ω)
7: **`LoadTerminatorDetectorKMs`**(Ω)
8: **for** each system call i **in** $\mathcal{T}$ **do**
9:     **for** each RAT j **in** $\mathcal{T}$ **do**
10:         t'$_{ij}$ $\leftarrow$ **`MeasureSyscallRAT`**($\mathcal{A}$, Ω)
11:         **if** t$_{ij} \neq$ t'$_{ij}$ **then**
12:             $\mathcal{B} \leftarrow \mathcal{B} + 1$
13:             $\mathcal{C} \leftarrow$ **`CollectTraces`**(t'$_{ij}$)
14:         **end if**
15:         **`RestoreAppInitialState`**($\mathcal{A}$)
16:     **end for**
17: **end for**
18: **`UnloadTerminatorDetectorKMs`**(Ω)
19: **return** $\mathcal{B}$, $\mathcal{C}$

regular terminations of a cloud-based application, t_{ij} is the value of RAT j during the execution of a system call i when a cloud-based application is regularly terminated. T' is another matrix of size $n \times m$ for irregular terminations of a cloud-based application, t'_{ij} is the value of RAT j during the execution of a system call i when a cloud-based application is irregularly terminated. D is the delta function that evaluates the presence of BASIR by comparing the difference between the value of RAT when a cloud-based application is regularly terminated and the value of the same RAT when this application is irregularly terminated during the execution of the same system call. B is the summation function that computes the total number of BASIR by analyzing executions between irregular and regular terminations of a cloud-based application for m RAT and n system calls.

`T-BASIR` is illustrated in Algorithm 1 that contains the following main phases: (i) send termination signals to certain system calls of a cloud-based application, and (ii) measure the impacts on the state of RAT when the cloud-based application is irregularly terminated during the execution of these system calls. The algorithm for `T-BASIR` takes in the entire set of inputs for the cloud-based application, its snapshot, and the KM configurations Ω, containing the identifier, terminator and detector KMs. Starting from Step 2, the algorithm loads the identifier KM into an operating system. In `T-BASIR`, we use lock system calls, where a thread locks certain resources to perform read or write operations. In Steps 3-5, the identifier KM randomly selects a set of system calls and records the values of RAT that are used by these system calls when the cloud-based application is regularly terminated. In Step 6, the identifier KM is unloaded from the operating system. In Step 7, the terminator and the detector KMs are loaded into the operating system. In Steps 8-17, for each system call and RAT, the algorithm repeatedly runs the snapshot of the cloud-based application, and then the terminator KM sends a termination signal to the cloud-based application during the execution of this system call. For each run, the detector KM uses Eq. (2) to measure the difference between the value of RAT when the cloud-based application is regularly terminated and the value of the same RAT when this application is irregularly terminated during the execution of the same system call. Once a difference is found, the detector KM uses Eq. (1) to add this difference to the total number of potential BASIR and collects its traces, as discussed in Section III-C. The cycle of Steps 8-17 repeats until the set of system calls is completed. Finally, in Step 18, the terminator and the detector KMs are unloaded from the operating system. The found instances of BASIR and their traces are returned in Step 19 as the algorithm ends.

C. *Identifying the Causes of BASIR*

Our goal is to automatically determine specific instructions in the source code of cloud-based applications that lead to BASIR when these applications encounter irregular terminations. In order to contrast instructions that lead to BASIR, we rely on the stack trace approach that can be used to collect execution traces from the stack in the memory when a cloud-based application is irregularly terminated. The stack traces contain a sequence of method calls with corresponding instructions, which often represents the current point in the execution path. These traces are often difficult to capture because termination signals can be initiated at every point in the execution of a cloud-based application, leading to a significantly larger search space. Hence, existing tracing tools are not applicable to `T-BASIR` because the stack traces of cloud-based applications are gone as soon as these applications are terminated. However, our tracer KM in `T-BASIR` can intercept a termination signal before this signal is delivered to a cloud-based application because this application runs inside our tracer KM (i.e., this application runs on a child process of the tracer/parent process). Then, our tracer KM first generates and prints the stack traces of a cloud-based application and then delivers this signal to terminate this application. As a result, developers can use these traces to identify corresponding instructions in the source code that lead to instances of BASIR.

IV. Empirical Evaluation

In this evaluation section, we state our *Research Questions (RQs)*, illustrate subject applications, describe our methodology to evaluate `T-BASIR`, and outline threats to its validity.

RQ$_1$: How effective is `T-BASIR` compared to the random approach in finding more instances of BASIR?

RQ$_2$: How effective is `T-BASIR` in finding different types of BASIR?

RQ$_3$: Is `T-BASIR` more effective than the random approach in causing more impacts on the application behaviors?

TABLE I: Overview of the applications: their names followed by the versions of the applications, and the total number of accessed futexes and their system calls when these applications restart after regular terminations.

Application	Version	Futexes	Syscalls
MySQL	v5.7.25	58	132
Cassandra	v3.0.17	35	138
PostgreSQL	v10.6	3	5
CouchDB	v2.3.0	25	11920
MongoDB	v3.0.6	61	1201
Hbase	v2.1.2	53	808
Docker	v18.09.0	45	1583
Hadoop	v3.0.3	34	1716
ZooKeeper	v3.4.12	35	910
Hive	v2.1.1	32	874

A. Subject Applications

We evaluated `T-BASIR` on 10 open-source subject applications. An overview of the subject applications is shown in Table I. These applications are multithreaded, have high popularity indexes, come from different domains, and are written by different programmers. The synchronization mechanism of these applications relies on a futex system call [5], which is a fast user-space synchronization method that puts specific threads to sleep/wait or wakes waiting threads when specific conditions become true. Each critical section in these applications often uses certain futex variables that are stored in particular memory addresses and are used by multiple threads to access this critical section through futex system calls.

B. Methodology

For each application, we first use the Strace tool [6] to ensure that its synchronization mechanism relies on futex system calls. As discussed in Section III-B, `T-BASIR` analyzes the values of the RAT between regular and irregular terminations at the same point in the execution to identify BASIR. RATs are the logs of the subject applications, the logs of the Linux kernel, the number of accessed futexes, and the number of futex system calls. An application is irregularly terminated using the RANDOM approach, where a termination signal is sent to any point in the execution of this application, and in `T-BASIR`, where a termination signal is sent to specific points in the execution of this application (i.e., during the executions of futex system calls). `T-BASIR` uses the logs to identify different types of BASIRs that lead to different effects on the behaviors of applications to answer $\boldsymbol{RQ}_1$ and $\boldsymbol{RQ}_2$. `T-BASIR` also identifies other cases of BASIR when the logs do not contain error messages. For example, `T-BASIR` identifies when applications cannot restart without manual interventions using the process status tool [7]. Also, we measure the impacts on the behaviors of the subject applications to answer $\boldsymbol{RQ}_3$. When an application restarts after irregular terminations, we check if values for the total number of accessed futexes and their system calls vary from the expected values when this application restarts after regular terminations for 20 seconds, which is set experimentally. Once a significant change is identified, as discussed in Section III-B, `T-BASIR` adds this change to the total number of potential BASIR and collects its traces. `T-BASIR` is implemented using KMs, KProbe, and JProbe interfaces [1]. The experiments for the subject applications were carried out using 10 virtual machines. Each subject application was deployed on Ubuntu 18.04 LTS VM with 4 GB of memory and 4 GHz CPU. For each application, we created a snapshot to ensure a similar state of the test environment after irregular terminations.

C. Threats to Validity

Our implementation of `T-BASIR` deals with only futex system calls, whereas other applications may use different synchronization mechanisms (e.g., semaphore system calls). While this is a potential threat, it is unlikely a major threat, since `T-BASIR` can be adjusted to support other types of synchronization mechanisms. In order to use `T-BASIR` with other applications, the developer can change only the system call type in the KMs so that `T-BASIR` identifies other types of system calls.

We experimented with only synchronization system calls, whereas other types of system calls (e.g., information flow, creation, preparatory, and termination) could also result in different effects on the behaviors of applications when these applications are terminated during the execution of other types of system calls. In contrast, understanding the effect of different types of system calls on the behavior of the applications is beyond the scope of this empirical study and shall be considered in future studies.

V. Empirical Results

In this section, we discuss the experimental results to answer the RQs listed in Section IV.

A. Finding more instances of BASIR

The experimental results to answer $\boldsymbol{RQ}_1$ are shown in Table II and summarize the found instances of BASIR when the subject applications encounter irregular terminations using `T-BASIR` and RANDOM approaches. We focus on determining whether these applications restart without manual interventions after they are irregularly terminated using `T-BASIR` and RANDOM. The experimental results show that `T-BASIR` causes MySQL, CouchDB, MongoDB, HBase, Hadoop, and ZooKeeper not to restart without manual interventions, whereas the RANDOM approach causes only CouchDB to not restart without manual interventions. Our explanation is that the RANDOM approach was able to cause CouchDB not to restart without manual interventions, since CouchDB uses an extremely high number of futex system calls, as shown in Table I. Hence, the RANDOM approach may accidentally hit these futex system calls, resulting in an instance of BASIR.

On the other hand, `T-BASIR` was not able to cause PostgreSQL, Cassandra, Docker, and Hive not to restart without

TABLE II: The comparison of the results of BASIR for T-BASIR and RANDOM. The first column specifies the name of the subject applications followed by columns for T-BASIR and RANDOM, and the cells indicate whether irregular terminations using these approaches lead to BASIR (i.e., an application cannot restart without manual interventions).

Application	T-BASIR	RANDOM
MySQL	✓	✗
Cassandra	✗	✗
PostgreSQL	✗	✗
CouchDB	✓	✓
MongoDB	✓	✗
Hbase	✓	✗
Docker	✗	✗
Hadoop	✓	✗
ZooKeeper	✓	✗
Hive	✗	✗

manual interventions. Our explanation is that PostgreSQL uses an extremely low number of futex system calls as shown in Table I. This situation puts T-BASIR at a disadvantage to find BASIRs since causing BASIR often requires more interactions among threads that often occur when a large number of futex system calls are executed. Cassandra runs on Java processes using a Java Virtual Machine (JVM), and T-BASIR uses Java processes instead of the application name processes (i.e., Cassandra) to specify the desired process of an application for receiving termination signals. Subsequently, JVM may play some roles in reducing the effect on Cassandra since Cassandra receives termination signals through the JVM. Docker uses the resource isolation features for the kernel [1]. T-BASIR uses KMs to send termination signals to the process of the subject applications. Hence, these features may play some roles in reducing the effect on Docker when Docker receives termination signals. Even though the Hive server restarts after irregular terminations using T-BASIR, its HCatalog component fails to restart. This observation allows us to conclude that even though irregular terminations may not show an impact on the restart state of an application, it does not mean that the other components of this application have no impacts too. In summary, our results show that T-BASIR causes six subject applications not to restart without manual intervention, whereas the RANDOM approach causes only one subject application not to restart without manual intervention, thus **positively addressing *RQ*$_1$**.

B. Finding different types of BASIR

When we investigate ***RQ***$_2$, we observe that unlike the RANDOM approach, T-BASIR leads to other types of BASIR. Since we are more familiar with the MySQL components, we further analyze and discuss the effects of other types of BASIR for MySQL. We observe that the logs of MySQL report the following message. `[Note] InnoDB: page_cleaner: 1000ms intended loop took 848417ms` [2]. The

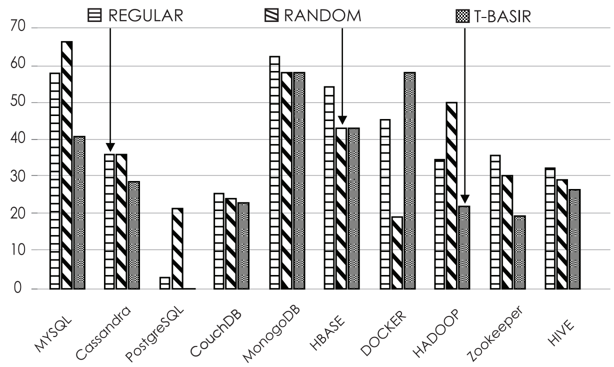

Fig. 2: Comparing the total number of accessed futexes when the subject applications restart after regular and irregular terminations using T-BASIR and RANDOM approaches.

message shows that the `page_cleaner` method that is responsible for writing data from memory into the disk takes a very long time from 1 second, which is expected, to 848 seconds (∼14 minutes). This result demonstrates a major problem, since it results in not only performance bottlenecks but also data loss. We analyze the effect of data loss by creating a virtual machine with 1 GB of memory, and we use `MySQLlap client` to perform large write operations (e.g., inserting hundreds of records) using multiple threads. We then load T-BASIR into the operating system to send the termination signals during the execution of these system calls. Interestingly, we observed that once MySQL restarts, the recently written data is lost. This bug is also reported on the following web page [8]. Also, we observed the following error message: `[ERROR] InnoDB: Unable to lock ./ibdata1 error: 11` [2]. The error message shows that T-BASIR prevents MySQL from performing a clean shutdown and hence results in locked ibdata1, which is a file that includes the shared tablespace containing the internal data of InnoDB. Unlike the RANDOM approach, T-BASIR also leads to other types of BASIR, such as performance bottlenecks, data loss, and locked resources. This result confirms that T-BASIR also results in different types of BASIR, compared to the RANDOM approach, thus **positively addressing *RQ*$_2$**. As a result, when irregular terminations result in BASIR, T-BASIR collects the traces that contain a sequence of method calls with corresponding instructions, as discussed in Section III-C. Hence, developers can use these traces to improve the design of the shutdown process for the subject applications.

C. Impact of T-BASIR on the behaviors of applications

The results of the experiments are presented in the histogram plot in Fig. 2 that summarizes the number of accessed futexes for the subject applications when these applications restart after regular and irregular terminations using T-BASIR and RANDOM approaches. These futexes often control the access of shared resources in critical sections across various threads/processes of an application. Different futexes often correspond to different execution paths since these futexes control the access of critical sections in different methods of an application. We observe that the number of accessed futexes varies between regular and irregular terminations using `T-BASIR` and RANDOM approaches. This observation suggests that the execution paths between regular and irregular terminations of an application change where newly accessed futexes (i.e., extra futexes) may have been accessed in the recovery execution paths, or other futexes that are often used during the execution of the application startup may not have been accessed (i.e., missing futexes). We observe that, except for Docker, most numbers of accessed futexes when applications are irregularly terminated using `T-BASIR` are lower than the number of accessed futexes when applications are regularly terminated or irregularly terminated using the RANDOM approach. A higher change in the number of accessed futexes often indicates a higher change in the execution paths when an application restarts after regular and irregular terminations. Further details about the results for all applications are shown in Fig. 3, where the number of extra and missing futexes are provided. Interestingly, we observe that there is a change in the number of accessed futexes between `T-BASIR` and RANDOM approaches, which suggests when an application encounters irregular terminations using different approaches, it often leads to different execution paths for the application. Hence, this observation confirms that the change in the execution paths not only indicates the recovery execution paths but also indicates other execution paths that may result in instances of BASIR. As a result, these experimental results demonstrate that when applications encounter irregular terminations using different approaches, it often leads to different execution paths, which result in different impacts on the behaviors of these applications.

To investigate $\boldsymbol{RQ}_3$ further, we present the change in the number of futex system calls for CouchDB in Table III when this application restarts after regular and irregular terminations using `T-BASIR` and RANDOM. Due to page limitations, we only present the results for CouchDB. The experimental results for other applications can be found in the online appendix [2]. We observe that the number of futex system calls when CouchDB is irregularly terminated using BASIR, except for a few futexes, is greater than the number of futex system calls when CouchDB is regularly terminated or irregularly terminated using the RANDOM approach. This result suggests that irregular terminations that are initiated by `T-BASIR` often lead to more impacts on the behaviors of applications compared to the RANDOM approach, since the higher number of futex system calls indicates not only more thread contentions but also a higher chance of locked resources. Interestingly, we observe that a futex with the memory address $0x0610$ has a significant decrease in the number of its futex system calls between regular and irregular terminations, which suggests some threads that use this futex may be prevented (i.e., locked) from reaching this point in the execution. In summary, these experimental results demonstrate that `T-BASIR` not only results in different impacts on the behaviors of these applications but also leads to more impacts on the behaviors of these applications compared to the RANDOM approach, thus **positively addressing** $\boldsymbol{RQ}_3$. As a result, when certain futexes result in significant changes in the behavior of applications, the traces of these futexes can be reviewed by developers to analyze how the changes of these futexes and their traces may lead to instances of BASIR.

TABLE III: The comparison of the total number of futex system calls for CouchDB after regular and irregular terminations. The first column designates the last four digits of the memory address for a futex. The following columns designate REGULAR, RANDOM, and `T-BASIR`, respectively.

Address	REGULAR	RANDOM	T-BASIR
0x12c8	3	0	0
0x0190	400	512	522
0x01d0	417	516	528
0x0210	396	518	526
0x0250	409	506	518
0x0290	414	522	530
0x02d0	412	512	528
0x0310	397	526	534
0x0350	402	518	528
0x0390	449	578	584
0x03d0	403	520	532
0x0410	405	522	538
0x0450	392	523	530
0x0490	563	705	686
0x04d0	396	520	528
0x0510	391	506	507
0x0550	382	514	522
0x0590	6	8	10
0x05d0	11	15	13
0x0610	5245	3402	3315
0xdf78	3	3	0
0xf7f8	3	6	3
0x95c8	19	5	9
0x95cc	1	1	1
0x9660	1	1	1

VI. Related Work

To the best of our knowledge, `T-BASIR` is the first automated solution for testing the effect of spot instance revocations on cloud-based applications. Most of the prior works focused on reducing the effect of spot instance revocations using fault-tolerance methods, such as replication [9]–[11], checkpointing [12]–[14], and VM migration [15], [16]. Voorsluys et al. [9] proposed a fault-aware resource allocation approach that applies the price of spot instances, runtime estimation of applications, and task duplication mechanisms to economically run batch jobs in spot instances. Yi et al. [13] proposed checkpointing schemes to reduce the computation price of spot instances and the completion time of tasks. Shastri et al. [16] proposed a resource container that enables applications to self-migrate to new spot VMs in a way that optimizes cost-efficiency as the spot prices change.

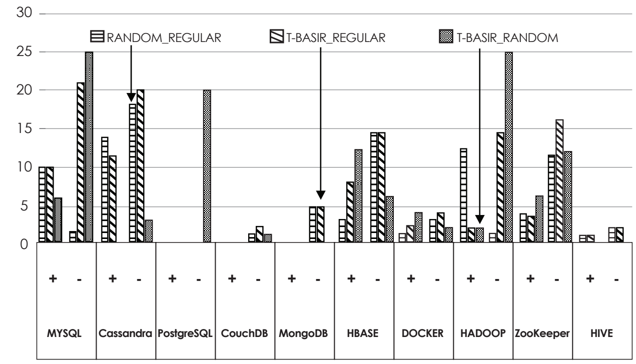

Fig. 3: The change in the total number of accessed futexes between regular and irregular terminations using `T-BASIR` and RANDOM approaches for the subject applications. The plus and minus symbols specify extra and missing futexes, respectively. The horizontal stripes, diagonal stripes, and dotted bars represent the change of accessed futexes between RANDOM and REGULAR, T-BASIR and REGULAR, and T-BASIR and RANDOM approaches, respectively.

In addition, other researchers worked on modeling spot markets to reduce the spot instance cost and the performance penalty that results from a high number of revocations, by designing optimal bidding strategies [17]–[20] and developing prediction schemes [21]–[23]. Song et al. [17] proposed an adaptive bidding approach that leverages the spot price history information to choose the bid strategy that increases the profit for brokers of the cloud service. Javadi et al. [22] proposed a statistical approach to analyze changes in spot price variations and the time between price variations to explore characterization of spot instances that are required to design fault-tolerant algorithms for applications deployed on cloud spot instances.

VII. Conclusion

We addressed a new and challenging problem for cloud-based applications that results from spot instance revocations. We proposed a novel solution to automatically find Bugs of cloud-based Applications that result from Spot instance Revocations (BASIR) and to locate their causes in the source code. We developed our solution for Testing the BASIR (`T-BASIR`), and we evaluated it using 10 popular open-source applications. The results show that `T-BASIR` finds more instances of BASIR and different types of BASIR, such as performance bottlenecks, data loss and locked resources, and applications that cannot restart, compared to the Random approach. With `T-BASIR`, developers can analyze the traces of BASIR to improve the design of the shutdown process for cloud-based applications and, hence, to gain the advantage of cloud spot instances in the cloud. This enables stakeholders to economically deploy their applications on the cloud spot instances. To the best of our knowledge, `T-BASIR` is the first automated solution to find and fix bugs of cloud-based applications resulting from spot instance revocations.

References

[1] J. Keniston, "The linux kernel documentation," https://www.kernel.org/, May 2019.

[2] "Our source code and experimental data," https://www.dropbox.com/s/0z4qndkv6dwkzhu/T-BASIR.zip?dl=0, May 2019.

[3] A. Alourani, M. A. N. Bikas, and M. Grechanik, "Search-based stress testing the elastic resource provisioning for cloud-based applications," in *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, 2018, pp. 149–165.

[4] D. Burleson, "Fix hanging shutdown: waiting for active calls to complete," http://www.dba-oracle.com/t_hanging_shutdown_waiting_-for_active_tasks_to_complete.htm, May 2019.

[5] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *AUUG Conference Proceedings*, vol. 85. AUUG, Inc. Kensington, NSW, Australia, 2002.

[6] P. Kranenburg, "Linux syscall tracer," https://strace.io, May 2019.

[7] W. E. Shotts Jr, *The Linux command line: a complete introduction*. No Starch Press, 2012.

[8] M. Mkel, "Move the innodb doublewrite buffer to flat files," https://jira.mariadb.org/browse/MDEV-11659, May 2019.

[9] W. Voorsluys and R. Buyya, "Reliable provisioning of spot instances for compute-intensive applications," in *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*. IEEE, 2012, pp. 542–549.

[10] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: agile ml elasticity through tiered reliability in dynamic resource markets," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 589–604.

[11] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy, "Spotcheck: Designing a derivative iaas cloud on the spot market," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015,

[12] S. Khatua and N. Mukherjee, "Application-centric resource provisioning for amazon ec2 spot instances," in *European Conference on Parallel Processing*. Springer, 2013, pp. 267–278.

[13] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 236–243.

[14] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy, "Spoton: a batch computing service for the spot market," in *Proceedings of the sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 329–341.

[15] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon, "Smart spot instances for the supercloud," in *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*. ACM, 2016, p. 5.

[16] S. Shastri and D. Irwin, "Hotspot: automated server hopping in cloud spot markets," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 493–505.

[17] Y. Song, M. Zafer, and K.-W. Lee, "Optimal bidding in spot instance market," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012.

[18] M. Zafer, Y. Song, and K.-W. Lee, "Optimal bids for spot vms in a cloud for deadline constrained jobs," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 75–82.

[19] S. Tang, J. Yuan, and X.-Y. Li, "Towards optimal bidding strategy for amazon ec2 cloud spot instance," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 91–98.

[20] S. Zaman and D. Grosu, "Efficient bidding for virtual machine instances in clouds," in *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 2011, pp. 41–48.

[21] R. Wolski, J. Brevik, R. Chard, and K. Chard, "Probabilistic guarantees of execution duration for amazon spot instances," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 18.

[22] B. Javadi, R. K. Thulasiramy, and R. Buyya, "Statistical modeling of spot instance prices in public cloud environments," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*. IEEE, 2011, pp. 219–228.

[23] C. Wang, Q. Liang, and B. Urgaonkar, "An empirical analysis of amazon ec2 spot instance features affecting cost-effective resource procurement," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, no. 2, p. 6, 2018.