T-BASIR: Finding Shutdown Bugs for Cloud-Based Applications in Cloud Spot Markets

Abdullah Alourani, *Student Member, IEEE*,
Ajay D. Kshemkalyani, *Senior Member, IEEE*, and Mark Grechanik, *Senior Member, IEEE*

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 improve the design of the shutdown process and is more efficient in finding instances of these bugs since it interposes at the system call layer.

Index Terms—Cloud computing, cloud spot markets, shutdown bugs of cloud-based applications, kernel modules, irregular terminations of cloud-based applications, spot instance revocations

1 Introduction

Cloud computing enables cloud customers to rent resources (e.g., virtual machines (VMs)) on as-needed basis to run their applications. That is, cloud customers do not have to buy and host expensive hardware to run their applications, and instead they rent resources for their applications from cloud computing facilities. This is an essential difference between cloud computing systems and distributed systems, which require application owners to buy and host expensive hardware to run their applications. As the deployment cost is an integral part of applications deployed on the cloud, the cost-efficiency of provisioning resource to these applications becomes a priority, and it is of growing significance, since the total spending that will be affected by cloud computing is over $1 trillion by 2020 [1].

Many cloud providers such as Amazon Web Services offer four types of instances (i.e., VMs) [2]: on-demand, reserved, dedicated, and spot (also known as preemptible). Cloud customers can pay for renting on-demand instances per hour without long-term commitments, and they cost the most. Also, they can rent reserved instances for a long term by making an upfront payment to cloud providers and thus pay a much lower rate than on-demand instances. A variation of reserved instances is a dedicated host, which is a physical server that is assigned only to a specific customer, and nobody besides this customer can use the resources of this host. Unlike the fixed-cost paying schemes mentioned above, a variable-cost paying scheme allows cloud customers to specify the price they are willing to pay for renting a spot instance to run their applications [2], and, depending on the varying demand from cloud customers, the price of this spot instance can go up if the demand increases and the number of available instances that can be supported by a finite number of physical resources in a data center of cloud providers decreases [3]. Conversely, the price of this spot instance can go down if the demand decreases and the number of available instances increases. If the customer's price is greater than the cloud provider's price that depends on the demand, a spot instance will be provisioned to customers' applications at the customer's price. However, when spot instances are already provisioned to customer applications and the cloud provider's price goes above the customer's price, the cloud providers will revoke those spot instances within two minutes by sending termination signals, thus resulting in *revocations of those spot instances* [3], whose occurrences are very difficult to predict [4]. As a result, even though cloud customers sometimes rent spot instances at 90 percent lower costs compared to on-demand [2], their applications that run in spot instances can be terminated based on price fluctuations that happen frequently, thus these applications may switch to an incorrect state leading to certain bugs [5], [6].

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

- *A. Alourani is with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, and also with Majmaah University, Al Majma'ah 15341, Saudi Arabia. E-mail: aalour2@uic.edu.*
- *A. D. Kshemkalyani and M. Grechanik are with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607. E-mail: {ajay, drmark}@uic.edu.*

Manuscript received 29 Oct. 2019; revised 19 Feb. 2020; accepted 7 Mar. 2020. Date of publication 13 Mar. 2020; date of current version 7 May 2020. (Corresponding author: Abdullah Alourani.)
Recommended for acceptance by Q. Zheng.
Digital Object Identifier no. 10.1109/TPDS.2020.2980265

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.

EC2 spot markets contain approximately 7,600 independent spot prices for different types of instances among 44 availability zones (i.e., data centers) in 16 regions [7]. 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. 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. In finer detail, when an application invokes synchronization system calls to lock a file and perform an update on the file inode's field that specifies if the file shall be persisted and this application is being irregularly terminated before the update is completed, system calls (i.e., fsync) of this application that are responsible for synchronizing the data of an open file to the storage device will become a "no-op", causing data loss of this file [8].

In general, heartbeat or timeout mechanisms might reduce the number of BASIR that require interaction between external processes (or threads) that run in different instances and an application's processes (or threads) run in a spot instance under irregular terminations, i.e., deadlocks, hangs, and performance bottlenecks. However, these mechanisms may not be useful for other types of BASIR that solely depend on ongoing calculations by applications deployed on a spot instance under irregular terminations, i.e., data corruption, data loss, crashes, and inconsistent states of shared data objects. Cloud-based applications that run in spot instances 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 this paper, 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) [9] to find these bugs and generate traces of their causes in the source code. `T-BASIR` is comprised of two major components. (1) Automating BASIR detection using KMs that contain the following main phases: (i) sending termination signals to certain system calls of a cloud-based application, and (ii) measuring the impacts on the state of RAT when the cloud-based application is irregularly terminated during the execution of these system calls. (2) Identifying the causes of BASIR using Tracer KM, which modifies the flow of executions through intercepting a termination signal to collect execution traces from the stack of a cloud-based application before the application receives the termination signal. BASIR and the traces of BASIR can be analyzed during application testing by developers, who look for fixes for BASIR to reduce or even eliminate the number of these bugs when cloud-based applications encounter irregular terminations. The motivation behind this work is to design a technique enabling developers to test the effect of spot instance revocations on cloud-based applications. The main contributions of this research work are:

- 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 bugs of applications resulting from cloud spot instance revocations.
- We evaluate `T-BASIR` using 10 popular open-source applications. Our results show that `T-BASIR` not only 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 improve the design of the shutdown process for cloud-based applications during their testing.
- `T-BASIR`'s code and our experimental results are publicly available [10], [11].

A preliminary version of these results appears in [12].

2 Related Work

In this section, we discuss the related work concerning spot instance revocations and application bugs.

2.1 Spot Instance Revocations

To the best of our knowledge, `T-BASIR` is the first automated solution for testing the effect of spot instance revocations on

TABLE 1
Comparison of T-BASIR With the Related Work Concerning Spot Instance Revocations and Application Bugs

(a) Spot Instance Revocations					
Modeling Spot Markets		Employing Fault-tolerance Mechanisms			Testing Impact on Applications
Bidding Strategies	Prediction Schemes	Replication	Checkpointing	Migration	
PADB [13]	DrAFTS [14]	Multifaceted Policy [15]	Spoton [16]	Smart Spot Instances [17]	T-BASIR
DBA [18]	Calibration [19]	Proteus [20]	Checkpointing Schemes [21]	Hotspot [22]	
AMAZING [23]	Quantitative Models [24]	Spotcheck [3]	ExoSphere [25]		

(b) Application Bugs			
Buggy Templates	Rules and Specifications	Historical Bugs	RAT
Metal Checkers [26]	Alattin [27]	HCM [28]	T-BASIR
PMD [29]	Pr-miner [30]	FixCache [31]	
FindBugs [32]	AFG [33]	Bug Prediction [34]	

The top part of Table (a) indicates existing works that aim to mitigate spot instance revocations. The following row designates the methodology of the proposed solution, followed by a row that designates specific methods. The bottom part of Table (b) indicates existing works that aim to find application bugs. The next row designates the methodology of the proposed solution and the cells contain the name of the proposed solutions.

cloud-based applications. Most of the prior works focused on reducing the effect of spot instance revocations using fault-tolerance methods, such as replication [3], [15], [20], checkpointing [16], [21], [25], and VM migration [17], [22]. Voorsluys *et al.* [15] 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.* [21] proposed checkpointing schemes to reduce the computation price of spot instances and the completion time of tasks. Shastri *et al.* [22] 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.

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 [13], [18], [23] and developing prediction schemes [14], [19], [24]. Song *et al.* [13] 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.* [19] 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.

2.2 Application Bugs

T-BASIR is the first automated solution to identify instances of BASIR. T-BASIR measures the impact on the state of RAT when the application is irregularly terminated to identify BASIR, as discussed in Section 4.3. Existing bug finding tools are not applicable to BASIR because they rely on searching through the application's execution paths for certain inputs to check if the state value of an application varies from the expected value that represents the input value of the next instruction in this execution path. However, a termination signal can be initiated at every execution state of applications, leading to a significantly larger search space of these states. Prior works required users to provide the buggy templates in order to find application bugs [26], [29], [32], whereas other works automatically inferred rules and specifications by mining existing applications in order to find application bugs [27], [30], [33]. Kermenek *et al.* [33] proposed a probabilistic approach that automatically infers specifications from a source code of an application and uses them to detect incorrect and missing properties in specifications. Other researchers focused on predicting application bugs using historical data of reported bugs [28], [31], [34]. Giger *et al.* [34] proposed a bug prediction approach that learns from source code and change metrics to predict application bugs.

2.2.1 Critical Analysis

In summary, Table 1 briefly gives a comparison of T-BASIR from different existing works that aim to mitigate spot instance revocations and find application bugs. While many of the prior works focused on reducing the effect of spot instance revocations by modeling spot markets and using fault-tolerance methods, these works are subject to altering pricing algorithms and are exposed to incurring overhead related to application completion time and deployment cost, respectively. In contrast, T-BASIR focuses on testing the effect of spot instance revocations on cloud-based applications. Also, although the other prior works focused on finding application bugs using buggy templates, rules and specifications, and historical bugs, these works are subject to limited inputs. However, T-BASIR measures the impact on the state of RAT when the application is irregularly terminated to identify BASIR, as discussed in Section 4.3. As a result, T-BASIR is the first tool that sheds light on the effect of spot instance revocations on cloud-based applications.

3 Problem Statement

In this section, we provide a background on shutdown processes and revocation notifications, discuss sources of BASIR, illustrate the BASIR problem, and formulate the problem statement.

3.1 Shutdown Processes and Revocation Notifications

The shutdown process of an application is often initiated during the execution of application instructions in response to

termination signals. This allows the application to switch its execution control to execute predefined shutdown instructions that save the state of the application and the state of its artifacts within a certain timeout before the operating system removes the application process from the memory. 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. Also, specifications describing the shutdown process of an application and which states are incorrect are rarely documented. The shutdown process of an application is often left untested because developers often assume that an application is properly terminated as long as its processes are terminated. As a result, the shutdown process of applications may fail to be completed within a certain timeout, leading to an incorrect state that affects the execution of subsequent instances of this application.

In general, cloud providers revoke (i.e., terminate) spot instances after a brief two-minute notification. The revocation notifications are often sent to spot instances when the demand from cloud customers 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. If the customer's price is greater than the cloud provider's price that depends on the demand, a spot instance will be provisioned to customers' applications at the customer's price. However, when spot instances are already provisioned to customer applications and the cloud provider's price goes above the customer's price, the cloud providers will revoke those spot instances within two minutes by sending termination signals [3]. The cloud providers give spot instances two-minute revocation notifications to enable applications that run in spot instances to be gracefully shut down within the two-minute revocation notice time. However, the brief two-minute revocation notice is not often enough to complete the shutdown process of applications, especially when the applications' memory footprints are greater than 4 GB [16]. As a result, when these applications are being terminated during the execution of the shutdown process of these applications, they might lose their states that lead to BASIR.

3.2 Sources of BASIR

There are two primary sources of BASIR. The first one is spot instance revocations. The other one is shutdown bugs of cloud-based applications.

3.2.1 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 [4]. 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 [1]. As a result, price fluctuations that depend on the demand have a high influence on the number of spot instance revocations.

In addition, it is very difficult for cloud customers to decide a price they are willing to pay for renting a spot instance to run their applications in such a way that reduces the deployment cost and the number of spot instance revocations [35]. When spot instances are already provisioned to cloud customer applications and the customer's price is close to zero, there is a high probability that those spot instances will be revoked by cloud providers. Also, when a cloud customer requests spot instances and the customer's price is close to zero, there is a very low probability that those spot instances will be provisioned to cloud customer applications. Conversely, if cloud customers set their prices close to on-demand instances' prices, cloud customers may reduce the number of revocations of spot instances that are provisioned to their applications, but cloud customers may not benefit from a significant discount of spot instances that is up to 90 percent compared to on-demand instances [2]. As a result, without knowing a demand from different cloud customers in advance, the challenge for cloud customers is to choose a price of spot instances that is both significantly lower than the price of on-demand instances and greater than the cloud provider's price to minimize the cost of the deployment and the number of spot instance revocations.

3.2.2 Shutdown Bugs of Cloud-Based 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. The shutdown sequence of an application is often left untested because developers often assume that an application is properly terminated as long as its processes are terminated. Developers often depend on the assumption that the operating system cleans the process space to a certain extent in any case. Also, specifications describing the shutdown process of an application and which states are incorrect are rarely documented. Unfortunately, existing bug finding tools (e.g., PMD [29] and FindBugs [32]) are not applicable to BASIR because they rely on searching through the application's execution paths for certain inputs to check if the state value of an application varies from the expected value that represents the input value of the next instruction in this execution path. However, a termination signal can be initiated at every execution state of applications, leading to a significantly larger search space of these states. On top of that, the shutdown sequence of an application varies based on the type of termination signals [9].

In addition, 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 [36]. 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

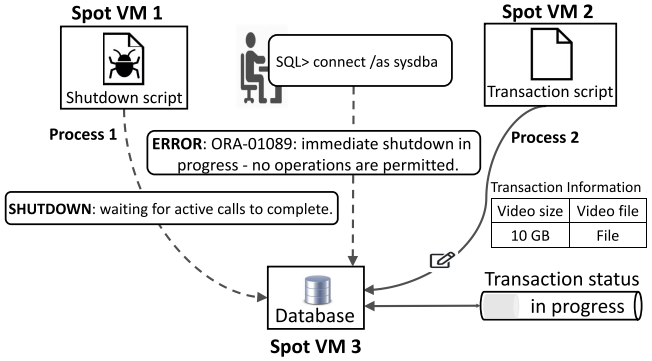

Fig. 1. An illustrative example of BASIR.

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.

3.3 Illustrative Example

The BASIR problem with a cloud-based application is illustrated in Fig. 1. As discussed in Section 3.2, 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 [5], 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., 10 GB), 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.

Additionally, we point out to multiple real-world bugs resulting from irregular terminations to shed light on the effect of spot instance revocations on applications. Irregular revocations could cause severe bugs, such as EX file system corruption [37], data loss on Atom editor [38], data loss on XFS file system [39], data corruption on Docker container [40], SQLite file corruption [41], database corruption on Docker [42], and Leveldb database corruption [43]. Also, the Linux documentations describe that although Linux can often repair file system corruption due to a power failure, some situations may require manual interventions to repair non-recoverable file system issues [44].

3.4 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 bugs of cloud-based applications that result from spot instance revocations*. Also, Eqs. (1) and (2) describe how to search through RAT for certain execution points (i.e., system calls) to check if the value t'_{ij} of RAT j during the execution of an execution point i when a cloud-based application is irregularly terminated varies from the expected value t_{ij} that represents the value of RAT j during the execution of an execution point i when a cloud-based application is regularly terminated. Once a difference b_{ij} is found, this difference is added to the matrix B of potential BASIR

$$B := T - T' \tag{1}$$

$$\begin{gathered} b_{ij} = \begin{cases} 0 & t_{ij} = t'_{ij} \\ (t_{ij} - t'_{ij}) & t_{ij} \neq t'_{ij} \end{cases} \\ \forall\, i \in \{1, \ldots, n\}, \forall\, j \in \{1, \ldots, m\}, \\ t_{ij} \in T, \quad t'_{ij} \in T', \quad b_{ij} \in B. \end{gathered} \tag{2}$$

Here, T is a matrix of size $n \times m$, n and m designate the total number of execution points (i.e., system calls) and RAT, respectively, for regular terminations of a cloud-based application, t_{ij} is the value of RAT j during the execution of an execution point (i.e., 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 an execution point i when a cloud-based application is irregularly terminated. Also, B is another matrix of size $n \times m$ for potential BASIR, b_{ij} is the difference between t_{ij} and t'_{ij}.

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, T-BASIR is the first automated solution to identify instances of BASIR. BASIR results from two primary sources: shutdown bugs of applications and spot instance revocations. However, since spot revocations are unpredictable and cloud-based applications are not designed or tested to deal with cloud spot revocations, BASIR is a critical problem for cloud customers, and T-BASIR is an essential tool to shed light on the effect of spot instance revocations on cloud-based applications. Thus, when the number of spot instance revocations or the number of shutdown bugs of applications increase, the number of BASIR will likely increase, and vice versa.

Specifically, we use kernel modules to find these bugs and generate traces of their causes in the source code. With our solution, developers can analyze the found bugs and their traces to improve the design of the shutdown process for cloud-based applications during the testing of these applications. Automatically finding these bugs is extremely difficult, in general, especially since a termination signal can be initiated at every execution state of applications, leading to a significantly larger search space.

4 Solution

In this section, we introduce KMs, explain why we use KMs, describe how we utilize KMs in T-BASIR and outline the architecture and workflow of T-BASIR.

4.1 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 [9]. 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) [9]. 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.

4.2 Why we use Synchronization System Calls in T-BASIR

In general, the synchronization system calls are responsible for managing the access of shared data objects among multiple processes (or threads). T-BASIR focuses on the synchronization system calls since the irregular terminations of synchronization system calls may negatively affect not only the state of shared data objects causing bugs (e.g., data loss, data corruption) but also the state of external processes (or threads) that run on different instances and interact with the process of terminated system calls, causing bugs (e.g., deadlocks and performance bottlenecks). However, although a write system call is another important type of system call that is responsible for modifying the value of data objects, the irregular termination of write system calls may negatively affect only the modified data objects, causing bugs (e.g., data loss, data corruption). Thus, the irregular termination of synchronization system calls may cause more bugs that are related to data objects and processes (or threads) within the critical section of synchronization system calls, compared to the irregular termination of the write system calls that may cause bugs related to only the modified data objects.

4.3 Automating BASIR Detection Using KMs

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 1. An essential goal is to identify which instructions of 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 [9] (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 [9] 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 6

$$B(T,T') = \sum_{i=1}^{n}\sum_{j=1}^{m} D(t_{ij},t'_{ij}) \text{ where } t \in T,\ \ t' \in T' \tag{3}$$

$$D\big(t_{ij},t'_{ij}\big) = \begin{cases} 0 & t_{ij} = t'_{ij} \\ 1 & t_{ij} \neq t'_{ij} \end{cases}. \tag{4}$$

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

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. (4) 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. (3) to add this difference to the total number of potential BASIR and collects the traces of this BASIR, as discussed in Section 4.4. As a result, with T-BASIR, developers can analyze the found instances of BASIR and their traces to improve the design of the shutdown process for cloud-based applications during the testing of these applications.

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** $\mathcal{B}$, $\mathcal{C}$

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. (4) 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. (3) to add this difference to the total number of potential BASIR and collects its traces, as discussed in Section 4.4. 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.

4.4 Identifying the Causes of BASIR

Tracer KM is at the core of the T-BASIR tracer to identify the causes of BASIR. We provide an overview and describe the implementation design of the T-BASIR tracer.

4.4.1 Overview of T-BASIR Tracer

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 [45] 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 [45] are not applicable to BASIR because the stack traces of 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, as discussed

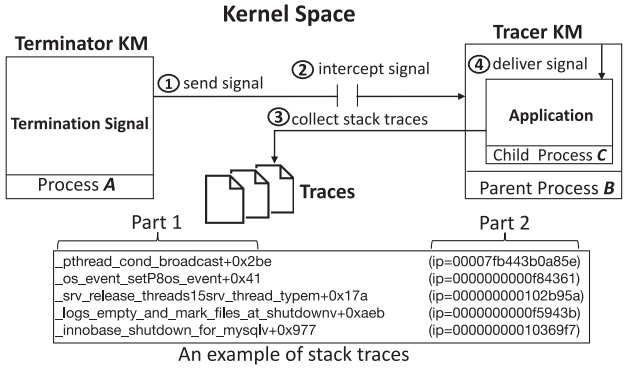

Fig. 2. The implementation design of T-BASIR tracer.

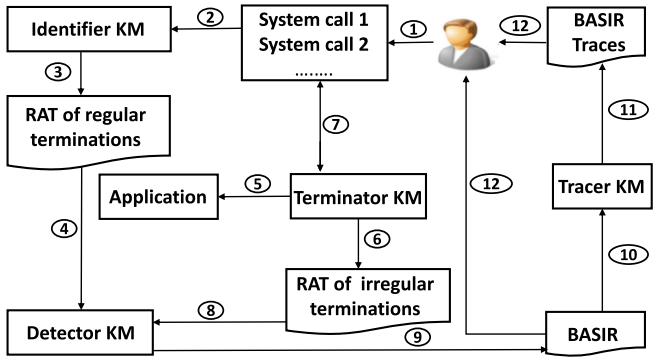

Fig. 3. The architecture and workflow of T-BASIR.

in Section 4.4.2. As a result, during application testing, developers can use these traces to identify corresponding instructions in the source code that lead to instances of BASIR.

4.4.2 Implementation Design of the T-BASIR Tracer

The implementation design of the T-BASIR tracer in the kernel space is illustrated in Fig. 2. The rectangles denote Terminator KM, Tracer KM, and a cloud-based application with their processes. The arrows indicate the actions between these KMs and the cloud-based application, and the numbers in the circles show the sequence of operations in the T-BASIR tracer.

Our tracer KM can intercept a termination signal before this signal is delivered to the cloud-based application because this application runs inside our tracer KM, as illustrated on the right side of Fig. 2 (i.e., this application runs on a child process of the tracer/parent process). In particular, when a termination signal is sent (1) to the cloud-based application, our tracer KM first intercepts (2) this signal to collect and store (3) the execution traces of the cloud-based application in files (e.g., text files) and then delivers (4) this signal to terminate this application.

In general, 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. Conversely, KMs have complete control over the execution in the kernel space at runtime. Hence, our tracer KM modifies the flow of executions through intercepting a termination signal to collect execution traces from the stack of a cloud-based application before the application receives the termination signal. In particular, our tracer KM uses Libunwind interfaces [46] to generate execution traces from the stack memory of the cloud-based application. An example of stack traces for a cloud-based application (e.g., MySQL) is illustrated in the bottom of Fig. 2. The first part of these traces refers to the sequence of method calls with corresponding instructions that represents the current point in the execution path when the cloud-based application is being irregularly terminated, and the second part of these traces refers to the methods' instruction pointers. As a result, the traces of BASIR can be reviewed by developers during application testing to identify which instructions in the execution path may lead to instances of BASIR.

4.5 T-BASIR's Architecture and Workflow

The architecture of T-BASIR is illustrated in Fig. 3. The rectangles indicate components of T-BASIR, the arrows denote the data flow between components, and the numbers in the circles show the sequence of processes in the workflow.

The input to T-BASIR is the entire set of inputs for a cloud-based application that performs specific operations (e.g., write) on certain resources (i.e., RAT), which often invoke particular system calls (e.g., acquire-lock) to use these resources. Initially, a set of system calls of the cloud-based application is chosen at random (1). For each system call, RATs are identified (2), and *Identifier KM* records (3) the values of RAT that are used by this system call when the cloud-based application is regularly terminated. These values of RAT that represent the expected values of the RAT, as discussed in Section 4.3, are passed (4) to *Detector KM*. *Terminator KM* sends (5) a termination signal to the cloud-based application during the execution of each system call. The values of RAT that are used by these system calls when the cloud-based application is irregularly terminated are collected (6). The evaluation is evolved using *Terminator KM* until the set of system calls is completed (7).

When the values of RAT for all system calls are collected, these values of RAT are passed (8) to *Detector KM*. *Detector KM* uses Eq. (4) to measure the difference between the value of RAT when the cloud-based application is irregularly terminated and the expected value of the same RAT when the cloud-based application is regularly terminated during the execution of the same system call. 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. Then, when a difference is found, *Detector KM* uses Eq. (3) to add this difference to the list of potential BASIR (9). Once the list of potential BASIR is obtained (10), *Tracer KM* collects (11) the traces of BASIR that contain a sequence of method calls with corresponding instructions, as discussed in Section 4.4. The found instances of BASIR and their traces are given to the developers for further evaluation (12).

5 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?

TABLE 2
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

TABLE 3
The Comparison of the Results of BASIR for T-BASIR and RANDOM

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

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?

5.1 Subject Applications

We evaluated T-BASIR on 10 open-source subject applications. An overview of the subject applications is shown in Table 2. 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 [47], 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 [47].

5.2 Methodology

For each application, we first use the Strace tool [9] to ensure that its synchronization mechanism relies on futex system calls. As discussed in Section 4.3, 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 RQ_1 and 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 [9]. Also, we measure the impacts on the behaviors of the subject applications to answer 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 4.3, 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 [9]. 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.

5.3 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 [9]). 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.

6 Empirical Results

In this section, we discuss the experimental results to answer the RQs listed in Section 5.

6.1 Finding More Instances of BASIR

The experimental results to answer RQ_1 are shown in Table 3 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 2. 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 manual interventions. Our explanation is that PostgreSQL uses an extremely low number of futex system calls as shown in Table 2. 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. 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* $\boldsymbol{RQ}_1$.

6.2 Finding Different Types of BASIR

When we investigate $\boldsymbol{RQ}_2$, 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: 1000 ms intended loop took 848417ms` [11]. The 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 [6]. Also, we observed the following error message: `[ERROR] InnoDB: Unable to lock ./ibdata1 error: 11` [11]. 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* $\boldsymbol{RQ}_2$. 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 4.4. Hence, developers can use these traces to improve the design of the shutdown process for the subject applications during the testing of these applications.

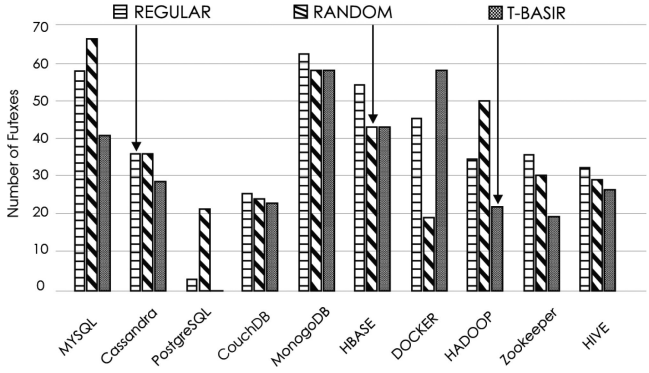

Fig. 4. The comparison of the total number of accessed futexes for regular and irregular terminations.

6.3 Impact of T-BASIR on the Behaviors of Applications

The results of the experiments are presented in the histogram plot in Fig. 4 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) [36]. 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

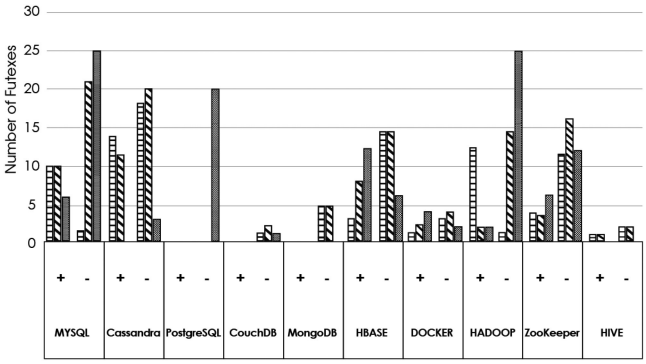

Fig. 5. The horizontal stripes, diagonal stripes, and dotted bars represent the change of accessed futexes between RANDOM and REGULAR, T-BASIR and REGULAR, and T-BASIR and RANDOM approaches, respectively. The plus and minus symbols specify extra and missing futexes, respectively.

after regular and irregular terminations. Further details about the results for all applications are shown in Fig. 5, 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 [36]. 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 4 when this application restarts after regular and irregular terminations using T-BASIR and RANDOM. The experimental results for other applications can be found in appendix (see the supplemental file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.2980265). We assume that running recovery execution paths of an application multiple times leads to the same values of the futex system calls for certain futexes in different runs. Hence, when the number of these futex system calls of an application after irregular termination using T-BASIR varies from the number of these futex system calls of this application after irregular termination using the RANDOM approach, it suggests the former recovery execution paths deviate from the latter recovery execution paths, which often indicates different impacts on the behaviors of this application. In particular, we observe that the number of futex system calls when CouchDB is irregularly terminated using T-BASIR, except for a few futexes, is often 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 last four digits of the memory address $0x0610$ for CouchDB 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.

TABLE 4
The Comparison of the Total Number of Futex System Calls for CouchDB After Regular and Irregular Terminations

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

Finally, we also observe that a futex with the last four digits of the memory address of $0x0020$ appears in extra futexes across different applications, such as Hadoop, HBase, and Hive, when they are restarted after irregular terminations. This observation suggests that this futex is invoked by recovery instructions of JVM, which is also reported on the collected traces of these applications [11]. Hence, fixing these recovery instructions of JVM will reduce or even eliminate the number of BASIR for all applications that rely on JVM. 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.

7 Conclusion and Future Work

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 during their testing 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 bugs of cloud-based applications resulting from spot instance revocations.

In the future, we plan to investigate the following research directions to extend our work.

- *Exploring the Impact of other Types of System Calls.* We plan to study the effect of I/O system calls that are responsible for reading/writing data from/to storage on applications when these applications are irregularly terminated during the execution of I/O system calls. For example, we will test the irregular termination of sync system calls that are responsible for synchronizing cached writes from volatile buffers to non-volatile buffers (i.e., persistent storage) to ensure that changes on volatile buffers are successfully flushed and committed to persistent storages on an irregular revocation. When sync system calls (i.e., fsync) are interrupted, due to irregular termination, we expect that cached writes in volatile buffers will likely not be committed to non-volatile storage, causing data loss. Another example of I/O system calls is write system calls. Let us suppose that concurrent write system calls are executed by separate processes/threads writing into a single buffer. However, consider what happens when one of these write system calls is interrupted by irregular termination. These processes/threads may not put all the data in a row, which causes data corruption.
- *Building Reliable Applications against Revocations.* We plan to build revocation-robust applications in cloud spot markets to reduce the number of BASIR when these applications encounter irregular terminations. In particular, our goal is to optimize the design of the shutdown sequence for these applications using certain specifications that describe the shutdown sequence. These specifications can be defined based on the developers' recommendations of the found instances of BASIR or common design flaws in the applications' shutdown process (i.e., bug reports in code repositories, as discussed in Section 3.3). For example, applications should make their buffered writes short (i.e., reducing the dirty data buffer time), and applications should first flush the primary data of applications in volatile buffers and then flush the secondary data of these applications (e.g., log files) in volatile buffers. Also, partitions used in applications should be mounted as read-only and temporary remounted as read and write during the write operations. However, if an irregular termination occurs during the write operations, partitions should be remounted as read-only, which will likely force flushing volatile-buffers faster, and additional writes to volatile buffers should be blocked. Hence, closing files that are opened for writing may reduce the negative effect on these files due to irregular terminations, whereas files that are open for reading will likely not be affected by irregular terminations. In general, applications should operate based on the magnitude of the termination interval to determine whether buffered writes can be stored in permanent stores, and they should also indicate whether the shutdown process is completed successfully. Finally, although such specifications cannot guarantee BASIR-free applications, they will likely reduce the number of BASIR when these applications encounter irregular terminations.
- *Comparing the Execution Time for Different Approaches.* We plan to present the comparison results of the execution time for T-BASIR and RANDOM approaches in future works. The comparison results of the execution time require us to set up 10 virtual machines for 10 subject applications and rerun all experiments for both approaches.

REFERENCES

[1] A. Alourani, M. A. N. Bikas, and M. Grechanik, "Search-based stress testing the elastic resource provisioning for cloud-based applications," in *Proc. 10th Int. Symp. Search-Based Softw. Eng.*, 2018, pp. 149–165. [Online]. Available: https://doi.org/10.1007/978-3-319-99241-9_7

[2] AWS, "Amazon EC2 spot instances," 2020. [Online]. Available: https://aws.amazon.com/ec2/spot/

[3] P. Sharma, S. Lee, T. Guo, D. E. Irwin, and P. J. Shenoy, "SpotCheck: Designing a derivative IaaS cloud on the spot market," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 16:1–16:15. [Online]. Available: https://doi.org/10.1145/2741948.2741953

[4] B. Sharma, R. K. Thulasiram, P. Thulasiraman, and R. Buyya, "Clabacus: A risk-adjusted cloud resources pricing model using financial option theory," *IEEE Trans. Cloud Comput.*, vol. 3, no. 3, pp. 332–344, Third Quarter 2015.

[5] D. Burleson, "Fix hanging shutdown: Waiting for active calls to complete," 2020. [Online]. Available: http://www.dba-oracle.com/t_hanging_shutdown_waiting_for_active_tasks_to_complete.htm

[6] M. Mäkelä, "Move the InnoDB doublewrite buffer to flat files," 2020. [Online]. Available: https://jira.mariadb.org/browse/MDEV-11659

[7] S. Shastri and D. E. Irwin, "Cloud index tracking: Enabling predictable costs in cloud spot markets," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 451–463. [Online]. Available: https://doi.org/10.1145/3267809.3267821

[8] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Finding crash-consistency bugs with bounded black-box crash testing," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 33–50. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/mohan

[9] J. Keniston, "The Linux kernel documentation," 2020. [Online]. Available: https://www.kernel.org/

[10] T-basir code on github, 2020. [Online]. Available: https://github.com/Abdullah-687/T-BASIR

[11] Our source code and experimental data, 2020. [Online]. Available: https://www.dropbox.com/s/0z4qndkv6dwkzhu/T-BASIR.zip?dl=0

[12] A. Alourani, A. D. Kshemkalyani, and M. Grechanik, "Testing for bugs of cloud-based applications resulting from spot instance revocations," in *Proc. 12th IEEE Int. Conf. Cloud Comput.*, 2019, pp. 243–250. [Online]. Available: https://doi.org/10.1109/CLOUD.2019.00050

[13] Y. Song, M. Zafer, and K.-W. Lee, "Optimal bidding in spot instance market," in *Proc. IEEE INFOCOM*, 2012, pp. 190–198.

[14] R. Wolski, J. Brevik, R. Chard, and K. Chard, "Probabilistic guarantees of execution duration for amazon spot instances," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 18.
[15] W. Voorsluys and R. Buyya, "Reliable provisioning of spot instances for compute-intensive applications," in *Proc. IEEE 26th Int. Conf. Adv. Inf. Netw. Appl.*, 2012, pp. 542–549.
[16] S. Subramanya, T. Guo, P. Sharma, D. E. Irwin, and P. J. Shenoy, "Spoton: A batch computing service for the spot market," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 329–341. [Online]. Available: https://doi.org/10.1145/2806777.2806851
[17] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon, "Smart spot instances for the supercloud," in *Proc. 3rd Workshop CrossCloud Infrastructures Platforms*, 2016, Art. no. 5.
[18] M. Zafer, Y. Song, and K.-W. Lee, "Optimal bids for spot VMs in a cloud for deadline constrained jobs," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 75–82.
[19] B. Javadi, R. K. Thulasiramy, and R. Buyya, "Statistical modeling of spot instance prices in public cloud environments," in *Proc. 4th IEEE Int. Conf. Utility Cloud Comput.*, 2011, pp. 219–228.
[20] 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 *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 589–604.
[21] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud," in *Proc. IEEE 3rd Int. Conf. Cloud Comput.*, 2010, pp. 236–243.
[22] S. Shastri and D. Irwin, "HotSpot: Automated server hopping in cloud spot markets," in *Proc. Symp. Cloud Comput.*, 2017, pp. 493–505.
[23] S. Tang, J. Yuan, and X.-Y. Li, "Towards optimal bidding strategy for amazon EC2 cloud spot instance," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 91–98.
[24] C. Wang, Q. Liang, and B. Urgaonkar, "An empirical analysis of amazon EC2 spot instance features affecting cost-effective resource procurement," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, 2018, Art. no. 6.
[25] P. Sharma, D. E. Irwin, and P. J. Shenoy, "Portfolio-driven resource management for transient cloud servers," in *Proc. ACM SIGMETRICS / Int. Conf. Meas. Model. Comput. Syst.*, 2017, Art. no. 59. [Online]. Available: https://doi.org/10.1145/3078505.3078511
[26] B. Chelf, D. R. Engler, and S. Hallem, "How to write system-specific, static checkers in metal," in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, 2002, pp. 51–60.
[27] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 283–294.
[28] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88.
[29] PMD, "An extensible cross-language static code analyzer," 2020. [Online]. Available: http://pmd.sourceforge.net
[30] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 306–315, 2005.
[31] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 489–498.
[32] FindBugs, "Findbugs documents and publications," 2020. [Online]. Available: http://findbugs.sourceforge.net/publications.html
[33] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 161–176.
[34] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2012, pp. 171–180.
[35] Q. Liang, C. Wang, and B. Urgaonkar, "Spot characterization: What are the right features to model," in *Proc. Int. Workshop Syst. Analytics Characterization*, 2016. [Online]. Available: https://sites.google.com/site/sacconference2016/submission
[36] D. N. Armstrong, H. Kim, O. Mutlu, and Y. N. Patt, "Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery," in *Proc. 37th Annu. Int. Symp. Microarchit.*, 2004, pp. 119–128.
[37] Juniper, "Ex file system corruption," 2020. [Online]. Available: https://kb.juniper.net/InfoCenter/index?page=content&id=KB20570
[38] P. Musubi, "Data loss on atom editor," 2020. [Online]. Available: https://github.com/atom/atom/issues/11406
[39] D. Lebedev, "Data loss on XFS file system," 2020. [Online]. Available: https://superuser.com/questions/84257/xfs-and-loss-of-data-when-power-goes-down
[40] C. Patry, "Data corruption on docker container," 2020. [Online]. Available: https://github.com/scality/cloudserver/issues/662
[41] B. Duddridge, "SQLite file corruption," 2020. [Online]. Available: https://github.com/couchbase/couchbase-lite-ios/issues/1482
[42] R. Fay, "Database corruption on docker," 2020. [Online]. Available: https://github.com/drud/ddev/issues/748
[43] Benny, "LevelDB database corruption," 2020. [Online]. Available: https://github.com/google/leveldb/issues/733.
[44] S. Frampton, "File system corruption after power outage or system crash," 2020. [Online]. Available: https://www.tldp.org/LDP/lame/LAME/linux-admin-made-easy/crash-repair.html
[45] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 151–160.
[46] D. Watson, "Libunwind documentation," 2020. [Online]. Available: https://www.nongnu.org/libunwind/docs.html
[47] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in Linux," in *Proc. AUUG Conf.*, 2002, pp. 479–495.

Abdullah Alourani (Student Member, IEEE) received the bachelor's degree in computer science from Qassim University, Buraydah, Saudi Arabia, and the master's degree in computer science from DePaul University, Chicago, Illinois. He is currently working toward the PhD degree in the Department of Computer Science, University of Illinois at Chicago, Chicago, Illinois. His current research interests include the areas of cloud computing, distributed systems, and software engineering. He is a student member of the ACM.

Ajay D. Kshemkalyani (Senior Member, IEEE) received the BTech degree in computer science from the Indian Institute of Technology, Bombay, Mumbai, Maharashtra, India, in 1987, and the PhD degree in computer science from the Ohio State University, Columbus, Ohio, in 1991, respectively. He is currently a professor with the Department of Computer Science, University of Illinois at Chicago. His research interests include distributed computing, computer networks, and concurrent systems. In 1999, he received the US National Science Foundation Career Award. He has served on the editorial board of the *Elsevier Journal Computer Networks*, and the *IEEE Transactions on Parallel and Distributed Systems*. He has coauthored a book entitled *Distributed Computing: Principles, Algorithms, and Systems* (Cambridge University Press, 2011). He is a distinguished scientist of the ACM.

Mark Grechanik (Senior Member, IEEE) received the PhD degree in computer sciences from the University of Texas at Austin, Austin, Texas. He is currently an associate professor with the Department of Computer Science, University of Illinois at Chicago. He has served as a member of ACM SigSoft Executive Committee since 2009. His research area is software engineering in general, with particular interests in software testing, evolution, and reuse. He is also interested in problems that lie at the intersection of software engineering and data privacy. He has a unique blend of strong academic background and long-term industry experience. He is a senior member of the ACM.