TScope: Automatic Timeout Bug Identification for Server Systems

Jingzhu He, Ting Dai, Xiaohui Gu
North Carolina State University
{jhe16, tdai, xgu}@ncsu.edu

***Abstract*—Timeout is commonly used to handle unexpected failures in server systems. However, improper use of timeout can cause server systems to hang or experience performance degradation. In this paper, we present TScope, an automatic timeout bug identification tool for server systems. TScope leverages kernel-level system call tracing and machine learning based anomaly detection and feature extraction schemes to achieve timeout bug identification. TScope introduces a unique system call selection scheme to achieve higher accuracy than existing generic performance bug detection tools. We have implemented a prototype of TScope and conducted extensive experiments using 19 real-world server performance bugs, including 12 timeout bugs and 7 non-timeout performance bugs. The experimental results show that TScope correctly classifies 18 out of 19 bugs. Compared to existing runtime bug detection schemes, TScope reduces the average false positive rate from 47.24% to 0.8%. TScope is light-weight and does not require application instrumentation, which makes it practical for production server performance bug identification.**

I. Introduction

Timeout is commonly used as a failover mechanism in complex server systems. For example, when a server component s_1 sends a request to another component s_2, s_1 can use the timeout mechanism to avoid infinite waiting in case s_2 fails to respond. However, recent studies [1], [2], [3] show that timeout bugs widely exist in real world server systems and can cause server to hang or slowdown. As an example of real world production system failure, a timeout bug caused the Amazon DynamoDB server to experience a five-hour service outage in 2015 [4]. Furthermore, our previous detailed timeout bug study [3] shows that 80% timeout bugs produce no error message or misleading error messages, which makes them difficult to identify.

A. Motivating example

We use Hadoop-11252 [5] as a motivating example to illustrate how timeout bugs happen. This bug occurs in Hadoop common, a standard utility library for configuring Hadoop cluster. This timeout bug is caused by missing timeout checking for the remote procedure call (RPC) connection between different nodes. In Hadoop clusters, nodes communicate with each other using RPC connections. In the bug shown by Figure 1, when the RPC connection from the Secondary NameNode to the DataNode is broken, the Secondary NameNode should send a TCP reset message (i.e., the RST message) to inform the DataNode to close the connection. However, in the case of a power outage occurred on the Secondary NameNode, the

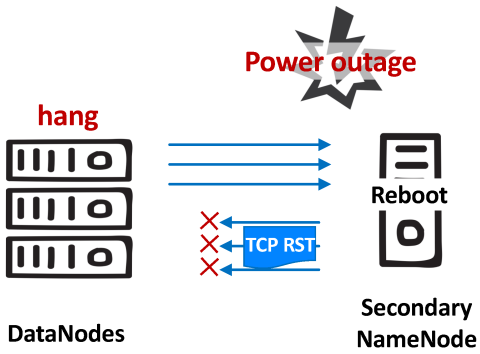

Fig. 1: Root cause of Hadoop-11252 bug. (The DataNodes miss timeout on RPC connection, leading to system hanging. Hadoop system reports no error message.)

RST messages get lost and the DataNode keeps its connection open forever and waits endlessly for the response from the Secondary NameNode even after the Secondary NameNode recovers from the power outage and reboots. As a result, the whole Hadoop system hangs and no error message is produced. To fix the bug, the developer added a one-minute timeout on the RPC connection from the DataNode to the Secondary NameNode. So the DataNode will close the RPC connection after it waits for a response from the Secondary NameNode for one minute. After timeout, the DataNode reconnects to the Secondary NameNode by establishing a new RPC connection with the Secondary NameNode after it recovers from the power outage.

B. Our Contribution

In this paper, we present TScope, a runtime timeout bug identification tool for detecting and classifying timeout bugs in server systems. When a server system experiences software hang or performance slowdown[1], TScope is triggered to identify whether the server performance anomaly is caused by a timeout bug. To be practical for production servers, TScope leverages kernel level system call tracing [8] to collect runtime system behaviors and performs bug identification based on the system call traces only. As a result, TScope does not require application source code or any application instrumentation.

[1]Those performance anomalies can be detected by various online anomaly detection tools (e.g., [6], [7])

Different from previous generic production server performance bug detection tools [9], TScope focuses on identifying timeout bugs for higher bug identification precision. To do so, TScope first introduces a unique feature selection scheme that performs filtering on the system call trace based on whether the system call is related to the timeout problem. TScope decides whether a system call is related to timeout based on three criteria: 1) a system call includes timeout related parameters (e.g., `timeout_msecs` in `sys_poll` system call); 2) a system call is related to network or synchronization; or 3) a system call is used in timeout configuration functions. The rationale for the first selection criteria is intuitive since a system call including a timeout related parameter will be likely invoked in the timeout mechanism with a high chance. The second selection criteria is derived from our previous timeout bug study [3] where we found many timeout bugs are triggered during network or synchronization operations. This observation is also expected since timeout is used for handling failures during inter-component communications or coordinations. The third criteria allows us to include those system calls that are used by timeout operations but not necessarily include timeout related parameters.

After selecting timeout related system calls, TScope extracts a list of feature vectors consisting of total execution time values of different system call types during each sampling interval (e.g., 1 second) from the raw system call traces. The execution time value of each system call type (e.g., `sys_poll`) reflects how long the system call gets executed during the sampling interval. TScope then performs multivariate anomaly detection using an unsupervised behavior learning method [6]. If any anomalies are detected, TScope then check whether those anomalies involve any system calls that include timeout related parameters. If the detected anomalies indeed include those timeout specific system calls, TScope infers the detected bug is a timeout related bug. Specifically, this paper makes the following contributions.

- We present a specialized timeout bug identification tool that combines timeout related feature selection and unsupervised machine learning based anomaly detection to achieve higher detection accuracy and more precise bug identification than previous generic bug detection tools.
- We introduce a timeout related system call selection scheme that comprehensively considers the system call parameters, where the system calls are invoked (e.g., network/synchronization operations), and when the system calls are invoked (e.g., during timeout configuration) to cover all the system calls that might be related to timeout bug identification.
- We have implemented a prototype of TScope and conducted extensive evaluation using 19 real-world performance bugs (12 timeout bugs and 7 non-timeout bugs) reported on 10 popular server systems. Our results show that TScope provides correct timeout bug identifications for 18 out of 19 performance bugs while 17 out of those 19 performance bugs produce no error message or misleading error messages. Compared to existing generic performance bug detection tools (e.g., PerfScope [9]), TScope can reduce the average false positive rate from 47% to 0.8%. TScope imposes negligible overhead to production server runtime executions and can produce timeout bug identification results in minutes.

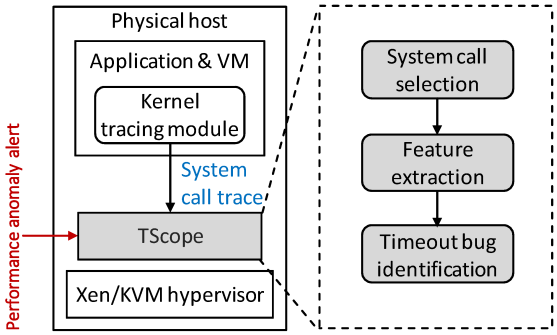

Fig. 2: The overall architecture of TScope.

The rest of the paper is organized as follows. Section II describes design details of TScope. Section III presents the experimental evaluation. Section IV discusses related work. Finally, the paper concludes in Section V.

II. System Design

In this section, we present the design details of the TScope system. We first provide an overview about TScope. We then describe the system call selection scheme followed by the timeout bug identification details.

A. Approach Overview

TScope consists four major components as shown by Figure 2. When a server system experiences software hang or performance slowdown, TScope first retrieves a window of system call trace from the kernel tracing module LTTng [8]. We chose LTTng because it incurs negligible overhead to the server system. In contrast, other system call tracing tools such as KProbes [10], DProbes [11] and SystemTap [12] often impose high overhead to the system. The output of LTTng contains each system call's timestamp, parameters, and other detailed system call information. Normally, an application can generate tens of millions of system call records per minute, which often provides sufficient data for analyzing performance bugs. Next, TScope performs system call selection to extract those system calls which are related to timeout issues. Third, TScope performs feature extraction over the raw system call trace to extract a list of feature vectors consisting of total execution time values of those selected system calls during each sampling interval. Next, TScope performs anomaly detection over the feature vectors to identify anomalous system calls and check whether those anomalous system calls are timeout related in order to infer whether the identified bug is a timeout bug.

B. System call selection

Linux provides over 300 system calls for users to use various privileged kernel functions. As mentioned in the Introduction, TScope performs timeout related system call selections using three criteria: 1) system calls that include timeout related parameters, 2) system calls that are related to network communications or synchronizations, and 3) system calls that are used by timeout configuration functions. We now discuss those selection strategies in detail.

System calls with timeout related parameters. We manually examine all the Linux system calls [13] and discover all of those system calls that contain timeout related parameters. For example, `sys_select` contains a timeout argument to determine how long a program should wait for files to become ready for I/O operations. During our previous timeout bug study [3], we observe that those system calls are often directly invoked when timeout bugs are triggered.

System calls related to network and synchronization. Similar to the first selection step, we manually extract all the system calls which are used by network communication or synchronization. For example, `sys_connect` is used to connects a socket to a specified address and `sys_fsync` is called for synchronizing a file's state with storage devices. The rationale for this selection criteria is that timeout is widely used for handling failures during inter-component communications or synchronizations. So we observe that many timeout bugs are triggered during network or synchronization operations [3]. As a result, those system calls related to network and synchronizations are likely to provide important hints for us to identify timeout bugs.

System calls used by timeout configuration functions. In this step, TScope tries to identify those system calls that are used by timeout configuration functions. To do so, TScope checks those functions which provide timeout configurations in standard C or Java libraries. For example, in Java 8 standard library, we check all the packages and identify those functions that provide timeout configurations such as `wait()` in `java.lang.Object` package that makes the current thread to delay execution for a certain time period. It contains a timeout parameter for users to define the maximum interval the thread should wait. Other examples include `sleep()` and `join()` in `java.lang.Thread` package, which define the sleeping time and maximum alive time of a thread, respectively. To collect system calls provided by those timeout configuration functions, we write simple test cases to run those functions and use LTTng to collect the system calls produced by those functions.

Using the above three selection criteria, TScope selects 125 system calls for timeout bug identification. Although we cannot guarantee that our scheme can include all the system calls that are related to timeout, we believe that the system calls we select are all highly correlated to real-world timeout bugs.

TABLE I: An example of how to calculate time vectors from processed system call lists.

System call	Entry timestamp	Exit timestamp
sys_socket	1523750400	1523750500
sys_read	1523750476	1523750542
sys_socket	1523750502	1523752040
sys_exit	1523751056	1523752037

C. System Call Feature Extraction

Each system call entry in the system call trace collected by LTTng consists a pair of records, i.e., `syscall_entry` and `syscall_exit` shown by Figure 3. Each record contains the timestamp when the system call occurs, the detailed system information, e.g., CPU ID, process ID and thread ID, and the system call's parameters. Some system calls may contain timeout arguments, e.g., `timeout_msecs` in `syscall_entry_poll`. We find that application-level timeout values are usually passed into these arguments. After we obtain system call trace using LTTng, we first extract the system calls occurred in the specific system using `procname`. For example, if we collect system calls for MySQL server system, we extract the system calls with `procname="mysqld"`. We then perform system call filtering based on the timeout related system call set derived by the previous step.

Next, we extract system call name, system call entry timestamp, thread ID and system call exit timestamp for each selected system call. System call name and thread ID are easy to extract directly. To determine the exact exit timestamp for each system call, we group system calls into different lists according to the thread ID. After we sort the system call list in each thread, the entry timestamp is determined by `syscall_entry` and the nearest corresponding `syscall_exit` of the same system call determines the exit timestamp. After exit timestamps are extracted, we sort all the system calls based on their entry timestamps.

We then segment the selected system call list into different samples based on a certain sampling interval (e.g., 1 second). To create feature vector for each sample, we use execution time of system calls. The rationale of choosing execution time over other features such as frequency is that we aim at identifying timeout bugs and timeout bugs often cause anomalies in system call execution time. Specifically, we extract a time vector $V = [x_1, ..., x_n]$ for each sample where x_i denotes the total execution time of all the occurrences of a system call s_i which appeared in the sample. For example, in Table I, `sys_socket` appears twice with the execution time values of 100 and 1538 milliseconds. So in the time vector of this sample, the value of `sys_socket` is set to $100 + 1538 = 1638$ milliseconds. The `sys_read` and sys_exit have the execution time of 66 and 981 milliseconds, respectively. So the final time vector for this sample is [1638, 66, 981] corresponding to three selected system calls of [`sys_socket`, `sys_read`, `sys_exit`]. Since we consider 125 different system calls, the time vector will have 125 dimensions. If a system call type

```
[14:24:43.520759222] syscall_entry_read: {cpu_id=...}, {..., pid=5004, ..., tid=5038}, {fd=3, ...}
[14:24:43.520759222] syscall_exit_read: {cpu_id=...}, {..., pid=5004, ..., tid=5038}, {ret = 30, ...}
[14:24:43.520760005] syscall_entry_write: {cpu_id=...}, {..., pid=5004, ..., tid=5038}, {fd=5, ...}
[14:24:43.520760218] syscall_exit_write: {cpu_id=...}, {..., pid=5004, ..., tid=5038}, {ret=1, ...}
[14:24:43.520943737] syscall_entry_poll {cpu_id=...}, {..., pid=5004, ..., tid=5038}, {..., timeout_msecs=60000}
[14:24:43.520943940] syscall_exit_poll: {cpu_id=...}, {..., pid=5004, ..., tid=5038}, {ret = -516, ...}
```

Fig. 3: System call trace example collected by LTTng.

does not appear in one sample, its value is set to be 0.

D. Timeout Bug Identification

We now describe how TScope identifies a performance bug as a timeout bug. To do so, TScope first performs anomaly detection over extracted feature vectors to detect any system call execution time anomalies preceding the system hang or performance slowdown. If any anomalies are detected, TScope checks whether those pinpointed abnormal system calls are directly related to timeout, that is, whether the system call includes timeout related parameters.

TScope leverages an unsupervised behavior learning (UBL) [6] to achieve efficient anomaly detection over high dimensional datasets (i.e., 125 dimensions). By choosing unsupervised methods, TScope can perform online anomaly detection without labeled training data. UBL is built on top of Self-Organizing Map (SOM) learning methods, which is one type of artificial neural network. The original implementation of UBL processes system-level metrics such as CPU, memory to detect system-level anomalies. TScope adapts the model to process system call execution time vectors. Compared to other anomaly detection algorithms such as clustering-based methods, SOM can map a high dimensional space into a low dimensional space while preserving the topological properties of original data space, which makes it work well for high dimensional data. Moreover, SOM can achieve higher accuracy than other approaches [14] by performing multi-variate anomaly detection over high dimensional data.

We now describe how TScope performs anomaly detection using SOM. The SOM model consists of a set of neurons each of which is associated with a weight vector. The weight vector has the same dimension as the time vector, which is 125 dimensions. During the model training phase, the SOM model uses a competitive learning method to update all the neurons. When a new training sample arrives, SOM calculates the Euclidean distance from the training vector to all the neurons and select the best matching neuron that has the smallest distance to the training vector. The weight vectors of the best matching neuron and its neighbors are updated using the training vector. During the anomaly detection phase, SOM finds the best matching neuron for the input time vector in the same way as the training phase. To decide whether an input vector is abnormal or not, we define a neighborhood area size (NAS) for each neuron as the sum of the Euclidean distance from the neuron to a set of its neighbors. We derived a threshold value for the NAS metric based on a user defined percentile value. If the NAS value of the best matching neuron for the input vector exceeds the threshold, we say the input vector is abnormal. The intuition behind the approach is that normal neurons are trained together many times, which all have small NAS values. In contrast, abnormal neurons are rarely trained, which have large NAS values.

To perform online anomaly detection, TScope splits the system call feature vector list into two halves with equal sizes. The first half is used as training data to create the SOM model while the whole set is used for anomaly detection. Note that SOM is resilient to a small number of noises in the training data. So our training is still valid even if the training data consist of abnormal system calls as long as the abnormal system calls are rare compared to normal system calls.

In addition to detecting anomalies, SOM also identifies which system calls attribute to the detected feature vector anomaly. TScope then checks whether those identified system calls include timeout related parameters to classify the detected bug as a timeout bug or non-timeout bug. For example, the timeout bug MapReduce-5066 is caused by missing timeout setting. When the bug is triggered, we observe that the execution time of `sys_epoll_wait` significantly increases. We observe that -1 passes into the timeout parameter of the `sys_epoll_wait`, causing the `sys_epoll_wait` block indefinitely. As an example of non-timeout performance bug, Cassandra-5064 bug is caused by an incorrect return value. When the bug occurs, the system falls into an infinite loop. We observe that the `sys_sched_yield` is called continually because the system is doing context switches endlessly consuming 100% CPU resources. However, `sys_sched_yield` does not include any timeout related parameters.

Notice that TScope uses different selection criteria for anomaly detection and timeout bug classification. We choose to use an expanded set of system calls (i.e., system calls including timeout related parameters, system calls related to network/synchronization, system calls invoked during timeout configuration) during anomaly detection, since just considering system calls with timeout related parameters sometimes makes the training dataset too small to yield accurate anomaly detection results. Our experimental evaluation results show the problem of limited system call selections.

III. Experiment Evaluation

This section presents our experiment evaluation. We implement a prototype of TScope and conduct our experiment on a host which is equipped with a quad-core Xeon 2.53Ghz CPU along with 16GB memory and runs 64-bit Ubuntu 16.04. The

TABLE II: System description.

System	Setup Mode	Description
Hadoop	Distributed	The utilities and libraries for Hadoop modules
HDFS	Distributed	Hadoop distributed file system
MapReduce	Distributed	Hadoop big data processing framework
Cassandra	Distributed	Distributed database management system
Phoenix	Distributed	Parallel and relational database engine
MySQL	Distributed	Scalable database
Zookeeper	Standalone	Synchronization service
Flume	Standalone	Log data collection/aggregation /movement service
Tomcat	Standalone	Java servlet container
Apache	Standalone	HTTP server system

system call trace is collected using LTTng 2.0.1. We introduce the evaluation methodology and the identification results. At the end of this section, we give two detailed examples of how TScope helps diagnose the timeout bugs.

A. Evaluation methodology

In this subsection, we describe our 19 real world bug samples and system call trace collection.

1) Real world bug samples: We collect all the bugs from ten open source systems. All the systems' names, description and setup modes are listed in Table II. These systems vary from back-end to front-end applications, constituting typical representatives and providing a wide coverage of server systems. We set up six systems in distributed modes, to investigate timeout issues occurring on the communication among different nodes in distributed systems.

We firstly describe timeout bug collection. We reproduce 12 timeout bugs, which are collected from bug repositories, e.g., Apache JIRA [15] and Bugzila [16]. Each report contains detailed information, e.g., version number, target platform and system's log information. Our reproduced bugs have a wide coverage of root causes and system impacts in our previous timeout bug study [3]. We cover four categories of root causes, i.e., misused timeout value, missing timeout, improper timeout handling and clock drifting, which occupy top 95% root causes. We cover three categories of impacts, i.e., system unavailability, job failure and performance degradation, which occupy top 98% of impacts brought by timeout bugs. We list the bugs' description in Table III.

To evaluate our classification result on non-timeout bugs, we reproduce 7 non-timeout performance bugs. We list them in Table IV. These bugs can also cause system calls' anomaly, making it difficult to distinguish those anomalies raised from timeout bugs.

For each bug, we start LTTng to collect system call trace just when the application is started. After the application runs for two to three minutes normally, we trigger the bug and record the bug triggering time. Then the system continues to run for about two to three minutes and we end up collecting the system calls. We separate the dataset into two halves. We use the first half of the dataset, representing the data generated in the normal state of the system, to train the anomaly detection model. We try our best to run some workloads during normal run, with the goal to reduce the false positives caused by high workloads. The workloads are also listed in Table III.

2) Alternative approaches: To evaluate TScope's performance, we compare TScope to two SOM based approaches, an existing bug detection and diagnosis tool, i.e., PerfScope, and another clustering approach, i.e., the DBScan clustering.

SOM-all and SOM-parameter: SOM-all and SOM-parameter refer to the approaches of using SOM model to identify timeout bugs under different selection sets. For SOM-all approach, we consider all the system calls. For SOM-parameter, we select the system calls containing timeout related parameters only. Since SOM-all and SOM-parameter select different sets of system calls, the false positive rates are different.

PerfScope: PerfScope is a performance bug detection and diagnosis tool. To be mentioned, we extract the execution units as the samples. An execution units refer to the system call clusters generated by the same function. We divide the system calls based on thread ID. Besides that, we divide the system call list according to time gaps between two consecutive system calls. We define the time gap threshold as the mean + 2×standard deviation. If the interval between two consecutive system calls is larger than this value, we segment the trace between the two system calls. We use the appearance vector as the features to cluster similar samples [17]. The appearance vector has a similar form as the time vector. The difference is that it only contains boolean variables and they represent whether a system call occurs in a sample. To identify the abnormal samples within clusters, we use the frequency vector and the time vector as two metrics to perform the nearest neighbor algorithm. The time vector is the same as we use. The frequency vector represents how many times a system call appears in a sample. We calculate the Euclidean distance of each sample's frequency (time) vector to its nearest neighbor's within each cluster. We set the threshold as the mean + 2× standard deviation. If one sample's distance to the nearest neighbor is larger than the threshold considering either the frequency vector or time vector, the sample is identified as anomaly.

Clustering: We implement DBScan clustering to identify timeout bugs. The implementation of sampling and feature vector extraction is same as TScope. The advantage of DBScan is that it does not require the number of clusters as input. DBScan algorithm's learning result is sensitive to the parameter ϵ and minimal points. ϵ defines the radius of a cluster. It is the threshold of Euclidean distance from the point to the cluster center. The false positive is reduced with the decreasing of ϵ. To reduce the false positive rate to the minimum, we set the ϵ to 1. The minimum points refers to the minimal number of points to form a cluster. We change the minimum points and find that it influence the identification result little. In our experiment, we set the minimal points to 5.

TABLE III: Timeout bugs' description.

Bug ID	System version	Root cause	Impact	Workload
Hadoop-11252	v2.5.0	Timeout is missing for the RPC connection	Hang	Word count for 765MB file
Hadoop-11252	v2.6.4	Timeout is misconfigured for the RPC connection	Hang	Word count for 765MB file
HDFS-10223	v2.6.4	Timeout setting is ignored and timeout value is hardcoded to a large one	Slowdown	Word count for 765MB file
Phoenix-2496	v4.6.0	Timeout is missing for synchronization	Slowdown	Database queries
MapReduce-5066	v2.0.3-alpha	Timeout is missing when JobTracker calls a URL	Hang	Word count for 765MB file
Cassandra-7886	v2.1.9	Wrong timeout handling	Hang	Database queries
Flume-1842	v1.3.0	Timeout is not calculated correctly	Slowdown	Writing log events to a file repeatedly
Zookeeper-1366	v3.5.0	Clock drifting	Crash	Checking expired events
Tomcat-56684	v6.0.39	Timeout value is set too high when accepting socket connection	Hang	Website browsing
Flume-1819	v1.3.0	Timeout is missing for reading data	Slowdown	Writing log events to a file repeatedly
Flume-1316	v1.1.0	Timeout is misconfigured	Slowdown	Writing log events to a file repeatedly
MapReduce-5724	v2.3.0	Timeout is missing	Hang	Word count for 765MB file

TABLE IV: Non-timeout performance bugs' description.

Bug ID	System version	Root cause	Impact	Workload
Cassandra-5064	v1.2.0-beta	Incorrect return value handling causes an infinite loop	Hang	Database queries
Apache-37680	v2.0.55	Incorrect flag causes infinite loop	Hang	Website browsing
Tomcat-48827	v6.0.24	Error in validating empty tag	Job failure	Website browsing
Tomcat-53450	v7.0.28	Upgrade a read lock to a write lock wrongly	Hang	Website browsing
MapReduce-3738	v0.23.1	Wait for an atomic variable to be set endlessly	Hang	Word count for 765MB file
MySQL-65615	v5.6.5-m8	Truncating tables causes disk flushing	Slowdown	Sysbench (benchmark)
MYSQL-54332	v5.5.5-m3	Two threads are deadlocked due to a locked table	Hang	Sysbench (benchmark)

B. Results analysis

We evaluate the accuracy of TScope from detection result, the false positive rate and the classification result of timeout bugs and non-timeout bugs.

1) Accuracy: We use the whole system call trace as input of SOM model. When SOM model raises a alarm, we check whether the alarms come from the samples after the bug is triggered. If an anomaly is reported after the bug is triggered, then the bug is viewed as truly detected. We use the standard false positive rate A_F to measure the experiment results. The equations are given in Equation 1. N_{fp} is the number of false positives, which means that TScope raise a false alarm on the a normal sample. N_{tn} is the number of true negatives, which means that TScope do not raise an alarm and it is a normal sample actually. To make the four approaches comparable, we all use the number of time slots in the same system call trace to calculate the A_F. The division of time slots are the same as sampling of TScope. N_{fp} represents the number of time slots with the false alarms. Similarly, N_{tn} represents the number of normal time slots without alarms. To be mentioned, for PerfScope, the time slot is identified as false positive if one execution unit, occurring on the time slot, reports a false alarm.

$$A_F = \frac{N_{fp}}{N_{fp} + N_{tn}} \tag{1}$$

The detection results are shown in Table V and the false positive rates are listed in Fig. 4. We observe that TScope can achieve 100% detection, while PerfScope and clustering can detect 11 and 10 out of 12 timeout bugs correspondingly. However, considering the false positive rate, TScope outperforms PerfScope and clustering a lot. For TScope, the average false positive rate is around 1%, while PerfScope and clustering have average false positives of 50% and 15%. PerfScope is also based on clustering methods. The clustering methods do not work well to find patterns for time vectors. It can be explained by the curse of dimensionality problem in the machine learning field. In our case, the time vectors have 125 dimensions, representing 125 system calls in our selection set. For a per-second trace sample, we find usually only 10% of the 125 system call occur. For the remaining system calls, the corresponding time vector values are all set to 0. The time vectors constitute a high dimensional sparse matrix. When we use clustering methods, it is difficult to organize the data and analyze the correlation between them. Therefore, the false positive rate is high. Another reason of over 50% false positive for PerfScope is that, there exit some false positive execution units which last for the whole system running time. The consequence is that PerfScope reports false alarms on many time intervals, when the system is running normally.

As shown from the detection results, compared with SOM-all, using TScope reduces the false positive rate by over 30%. The false positive rate is generally decreasing with the decreasing of the number of system calls in the system call selection set. It is intuitive that the anomalies should be reduced when we consider less system calls. The detection result of SOM-parameter approach proves that our selection set is most appropriate. For SOM-parameter approach, although the false positive rate is generally lower, the detection result is very bad, with only 7 out of 12 bugs detected. The reason is that the set of system calls with timeout related parameters is

TABLE V: Detection result of TScope and four alternative approaches for timeout bugs.

Bug ID	PerfScope	Clustering	SOM-all	SOM-parameter	TScope
Hadoop-11252 (v2.5.0)	✓	✓	✓	✗	✓
Hadoop-11252 (v2.6.4)	✓	✓	✓	✗	✓
HDFS-10223	✓	✗	✓	✓	✓
Phoenix-2496	✗	✓	✓	✓	✓
MapReduce-5066	✓	✗	✓	✗	✓
Cassandra-7886	✓	✓	✓	✗	✓
Flume-1842	✓	✓	✓	✓	✓
Zookeeper-1366	✓	✓	✓	✓	✓
Tomcat-56684	✓	✓	✓	✓	✓
Flume-1819	✓	✓	✓	✗	✓
Flume-1316	✓	✓	✓	✓	✓
MapReduce-5724	✓	✓	✓	✓	✓

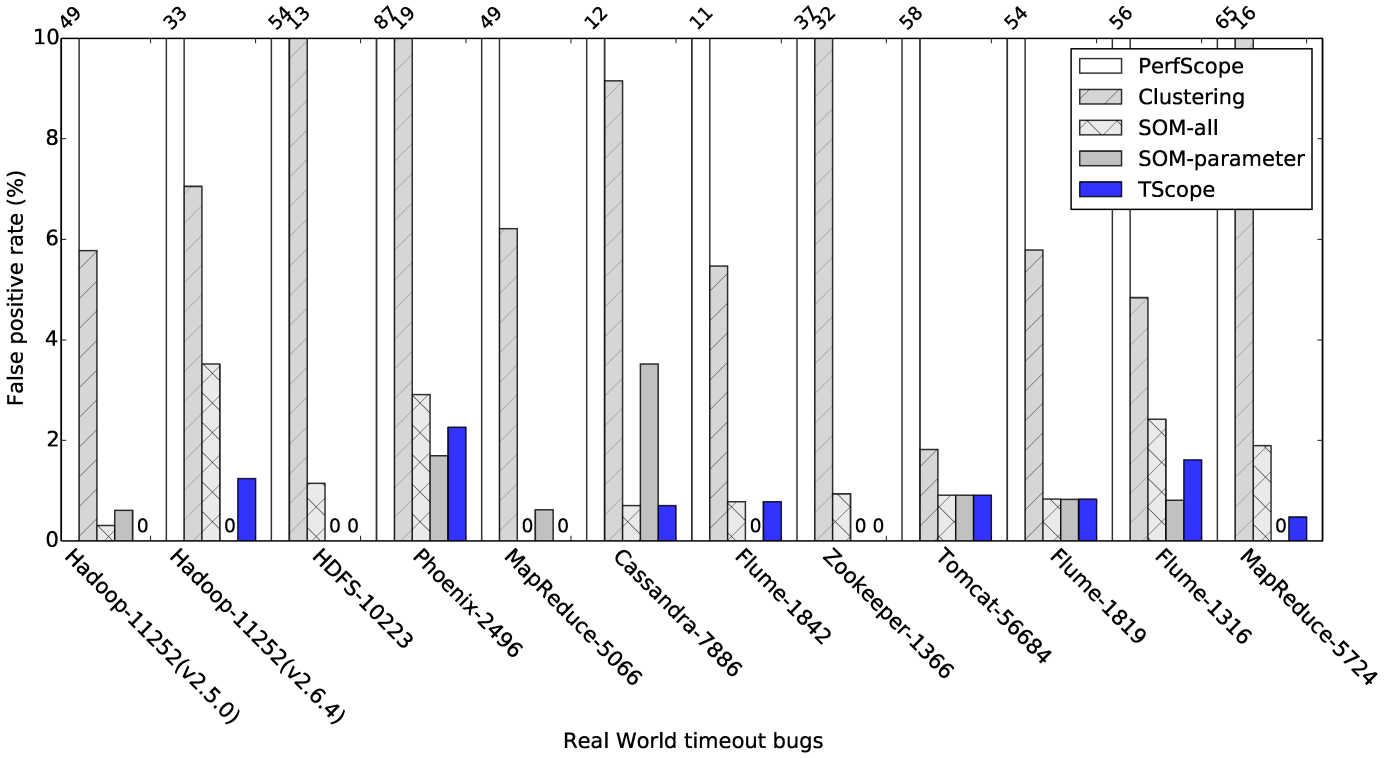

Fig. 4: False positive rate of TScope and four alternative approaches.

too small. Even though timeout value is passed into timeout related parameter, not all the anomalies caused by timeout bugs can manifest in these system calls. These system calls can call other system calls, further causing anomalies in those system calls. For example, in Flume-1819 bug, the execution time of `sys_wait4`, a system call without timeout parameter, is abnormally prolonged, while the system calls with timeout parameters do not report an anomaly. For Cassandra-7886, MapReduce-5066 and Hadoop-11252(v2.5.0) bugs, the false positive rates of SOM-parameter are even higher than TScope's. The reason is that there are only less than five system calls after selection for SOM-parameter approach, which makes training data too small to train a good model.

We conduct the classification experiments on 12 timeout bugs and 7 non-timeout performance bugs. The 7 non-timeout performance bugs cause either system hang or performance degradation. All the classification results are shown in Table VI. The results show that TScope can correctly classify 18 out of the 19 bugs. The one miss classification is Phoenix-2496 bug. This bug causes 10 seconds delay on the system, which is too short, compared with other timeout bugs causing system slowdown. TScope captures the short delay but it does not consider it as an anomaly caused by timeout bugs. TScope outperforms other four approaches in bug classification. For PerfScope and clustering approaches, the low classification accuracy is caused by the curse of dimensionality problem in high dimensional sparse datasets, which we have already discussed. SOM-all approach can identify 10 out of 12 timeout bugs and 3 out of 7 non-timeout bugs. SOM-all approach's low precision on non-timeout bugs is caused by no selection on system calls. SOM-parameter approach can only classify 10 out of the 19 bugs. It shows that the selection set of SOM-

TABLE VI: Classification result for the 19 bugs. No. 1 to 12 are timeout bugs, while No. 13 to 19 are non-timeout performance bugs. ✓ means the bug is identified as a timeout (non-timeout) bug and it is indeed a timeout (non-timeout) bug. ✗ means the bug is identified as a timeout (non-timeout) bug but it is a non-timeout (timeout) bug.

ID	Bug ID	Error Message	PerfScope	Clustering	SOM-all	SOM-parameter	TScope
1	Flume-1316 (Timeout)	Missing	✓	✗	✓	✓	✓
2	Flume-1819 (Timeout)	Missing	✓	✗	✓	✗	✓
3	MapReduce-5066 (Timeout)	Missing	✓	✗	✓	✗	✓
4	Hadoop-11252(v2.6.4) (Timeout)	Missing	✓	✗	✗	✗	✓
5	HDFS-10223 (Timeout)	Missing	✓	✗	✓	✓	✓
6	Tomcat-56684 (Timeout)	Missing	✓	✓	✓	✓	✓
7	Zookeeper-1366 (Timeout)	Missing	✓	✓	✓	✓	✓
8	Phoenix-2496 (Timeout)	Missing	✗	✗	✗	✓	✗
9	Hadoop-11252(v2.5.0) (Timeout)	Missing	✓	✓	✓	✗	✓
10	Cassandra-7886 (Timeout)	Misleading	✓	✓	✓	✗	✓
11	Flume-1842 (Timeout)	Missing	✓	✓	✓	✓	✓
12	MapReduce-5724 (Timeout)	Misleading	✓	✓	✓	✓	✓
13	Cassandra-5064 (Non-timeout)	Missing	✗	✗	✓	✗	✓
14	Apache-37680 (Non-timeout)	Missing	✗	✗	✓	✗	✓
15	Tomcat-48827 (Non-timeout)	Correct	✗	✗	✗	✗	✓
16	Tomcat-53450 (Non-timeout)	Correct	✗	✓	✓	✓	✓
17	MapReduce-3738 (Non-timeout)	Missing	✗	✗	✗	✗	✓
18	MySQL-65615 (Non-timeout)	Missing	✗	✗	✗	✓	✓
19	MySQL-54332 (Non-timeout)	Missing	✗	✗	✗	✓	✓

parameter cannot cover all the system calls related to timeout bugs. We can see that most of the bugs produce no error messages, even misleading messages, about the root causes. In this case, TScope provides useful guidance for developers to diagnose the bug. We introduce the Hadoop-11252(v2.6.4) case in detail in the next subsection. TScope can localize the buggy timeout variable for this bug.

Apart from checking the abnormal system calls reported by SOM model, there are another two reasons that TScope can classify timeout bugs. The first reason is that TScope uses time vectors to feed into the anomaly detection model. Our observation is that timeout bugs usually cause anomaly in system call's execution time, while non-timeout performance bugs can not. The manifestation of non-timeout bug is usually that the frequency of a particular system call's occurrence is changed or some rarely seen system calls occur when the bug is triggered. In our study, we also extract frequency vector, which represents how many times each system call appears in a sample. We find that the classification result is worse than that of using time vector. For example, Apache-37680 bug is mistakenly classified as timeout bugs using the frequency vector. The second reason is that TScope uses a unique system call selection strategy. Those anomalies that are not caused by timeout related system calls are filtered. For example, MySQL-54332 bug is mistakenly classified as timeout bugs before selection, while it is correctly classified as non-timeout bugs after selection.

2) Overhead: The LTTng tracing overhead is less than 1%, which is negligible. TScope does not require application profiling, which can impose significant overhead on systems. We list the computation time of log analysis on timeout bugs in Table VII. We can see that the average log size of the 12 timeout bugs is near 1000MB. Since the log size is large, it is significant to reduce the overhead to apply TScope in the real world cloud systems. In Table VII, feature extraction refers to the combination of sampling, system call selection, extracting the time vectors and training the anomaly detection model. The identification refers to using the built model to detect anomaly and further identify timeout bugs. The majority of the total computation time is spent on feature extraction. The reason is that traversing millions of system calls and categorizing them according to the system information is time consuming. We observe that the average computation time is tens to hundreds of seconds, which is fast enough to apply TScope in real-world cloud systems.

TABLE VII: Computation time of TScope on timeout bugs.

Bug ID	Log size (MB)	Feature Extraction (seconds)	Identification (seconds)
Hadoop-11252 (2.5.0)	1480	156.17	1.75
Hadoop-11252 (2.6.4)	1602	172.58	1.54
HDFS-10223	461	83.80	1.34
Phoenix-2496	710	37.79	2.04
MapReduce-5066	928	144.37	1.65
Cassandra-7886	355	70.75	1.64
Flume-1842	410	5.26	1.63
Zookeeper-1366	1206	112.58	1.52
Tomcat-56684	96	2.25	1.73
Flume-1819	463	8.05	1.65
Flume-1316	2687	26.24	2.62
MapReduce-5724	1304	96.31	2.21

C. Case Study

In this subsection, we discuss two examples in detail to show how TScope's identification results help to diagnose the timeout bugs.

1) Hadoop-11252(v2.6.4): The root cause of this bug is misconfiguring RPC timeout value on connection among Hadoop cluster nodes. The misconfigured timeout value is `Integer`'s maximum value, which is too long for the timeout value. When the bug occurs, the system hangs, producing no error message. TScope can detect the bug and identify it as a timeout bug. Besides, TScope reports five abnormal system calls, including the `sys_epoll_wait`. To find out the misconfigured value, we collect all the timeout related parameters of the five abnormal system calls. In this case, we can easily find that the `sys_epoll_wait` has an abnormal parameter, i.e., `Integer.MAX_VALUE`. Through matching all the timeout values of the corresponding variables in the Hadoop's `.xml` configuration files, we find that the misconfigured timeout value, i.e., `Integer.MAX_VALUE`, comes from the timeout variable `ipc.client.rpc-timeout.ms`, that exactly is the misconfigured timeout variable causing the bug.

2) Cassandra-7886: The root cause of this bug is missing timeout on the connection between the data node and the coordinator. The data node simply drops the data, when the input data is overwhelming. It does not inform the coordinator of the dropping data operation, causing the coordinator hanging on waiting for the acknowledgment response of successfully reading the data. The fixed version adds a timeout on the waiting response operation. When the bug occurs, the system reports the overwhelming exception, which is not relevant to the timeout mechanism. However, TScope detects the bug and identifies the bug as a timeout bug.

IV. Related Work

In this section, we discuss related work with a focus on describing the difference between Tscope and previous approaches.

Performance bug detection and diagnosis. Performance bug is notoriously difficult to detect and diagnose. Previous work has developed both static and dynamic analysis tools to address the challenge. For example, Fournier et al. [18] proposes to analyze dependencies among processes to understand how the total elapsed time is distributed among different processes. X-ray [19] presents performance summarization techniques to attribute performance costs to fine-grained events for diagnosing performance problems. PerfScope [9] uses system call tracing and frequent episode mining to localize root cause functions for performance anomalies. PerfCompass [20] presents a tool to differentiate external faults from internal faults based on the impact factor analysis over detected performance anomaly faults. Different from those generic performance bug detection and diagnosis tools, TScope focuses on identifying timeout problems that cause system hang or performance slowdown during production-run of server systems.

Machine learning based performance debugging. Work has been done to detect performance problems using machine learning techniques. Cohen et al. [21] presents a tool based on tree-augmented Bayesian networks to correlate system-level metrics with high-level performance states. Lee et al. [22] presents a fuzzy-prediction based self-tuning approach to improve the Hadoop system performance. Votke et al. [23] presents an analytical model to estimate performance under various interference conditions. EntomoModel [24] uses decision tree classification to depicts the workloads and management policies under which potential performance anomalies are likely to. UBL [6] leverages Self-Organizing Maps to capture emergent system behaviors and performs anomaly prediction for cloud systems. FChain [25] localizes faculty components based on the abnormal change propagation patterns and inter-component dependency relationships. Different from the above existing tools, TScope performs feature selection before applying machine learning algorithms to achieve high detection precision.

Static bug detection tools. There are a lot of existing work on developing static bug detection tools. Jin et al. [26] conduct a comprehensive study on performance bugs and propose efficiency rules to detect unknown bugs statically. DCatch [27] designs a set of happen-before rules to model concurrency mechanisms in distributed cloud systems. Other tools focus on detecting database's performance anomalies [28], sequential errors [29] and inefficient nested loops [30]. Different from those static analysis tools, TScope is a precise timeout bug identification tool. It identifies timeout anomalies by performing feature selection statically and anomaly detection dynamically.

V. Conclusion

In this paper, we have presented TScope, an automatic timeout bug identification tool for production server systems. TScope combines timeout related feature selection and runtime anomaly detection to achieve higher bug identification precision than previous generic performance bug detection tools. TScope does not require any application instrumentation for bug detection, which makes it practical for production server systems. We have implemented a prototype of TScope and conducted extensive experiments using 19 real world performance bugs. The experimental results show that TScope achieves much higher timeout bug identification accuracy than existing alternative schemes. TScope is light-weight and efficient, which imposes less than 1% runtime overhead to the production server and produces timeout bug identification results within minutes.

VI. Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This research is supported in part by NSF CNS1513942 grant and NSF CNS1149445 grant. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin *et al.*, "What bugs live in the cloud?: A study of 3000+ issues in cloud systems," in *SOCC*, 2014.

[2] J. Huang, X. Zhang, and K. Schwan, "Understanding issue correlations: a case study of the hadoop system," in *SOCC*, 2015.

[3] T. Dai, J. He, X. Gu, and S. Lu, "Understanding real world timeout problems in cloud server systems," in *IC2E*, 2018.

[4] "Irreversible Failures: Lessons from the DynamoDB Outage," http://blog.scalyr.com/2015/09/irreversible-failures-lessons-from-the-dynamodb-outage/.

[5] "Hadoop-11252," https://issues.apache.org/jira/browse/HADOOP-11252.

[6] D. J. Dean, H. Nguyen, and X. Gu, "UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *ICAC*, 2012.

[7] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "PREPARE: Predictive performance anomaly prevention for virtualized cloud systems," in *ICDCS*, 2012.

[8] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *Linux Symposium*, 2006.

[9] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang, "PerfScope: Practical online server performance bug inference in production cloud computing infrastructures," in *SOCC*, 2014.

[10] "KProbes," https://lwn.net/Articles/132196/.

[11] R. J. Moore, "A universal dynamic trace for linux and other operating systems." in *ATC*, 2001, pp. 297–308.

[12] "SystemTap," https://sourceware.org/systemtap/.

[13] "Linux system calls," http://man7.org/linux/man-pages/man2/syscalls.2.html.

[14] D. J. Dean, P. Wang, X. Gu, W. Enck, and G. Jin, "Automatic server hang bug diagnosis: Feasible reality or pipe dream?" in *ICAC*, 2015.

[15] "Apache JIRA," https://issues.apache.org/jira.

[16] "Bugzila," https://www.bugzilla.org.

[17] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009, vol. 344.

[18] P. Fournier and M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," in *SIGOPS*, 2010.

[19] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *OSDI*, 2012.

[20] D. J. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut, "Perfcompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds," *TPDS*, vol. 27, no. 6, pp. 1742–1755, 2016.

[21] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in *OSDI*, 2004.

[22] G. Lee and J. Fortes, "Hadoop performance self-tuning using a fuzzy-prediction approach," in *ICAC*, 2016.

[23] S. Votke, S. Javadi, and A. Gandhi, "Modeling and analysis of performance under interference in the cloud," in *MASCOTS*, 2017.

[24] C. Stewart, K. Shen, A. Iyengar, and J. Yin, "Entomomodel: Understanding and avoiding performance anomaly manifestations," in *MASCOTS*, 2010.

[25] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "FChain: Toward black-box online fault localization for cloud systems," in *ICDCS*, 2013.

[26] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.

[27] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "Dcatch: Automatically detecting distributed concurrency bugs in cloud systems," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 677–691.

[28] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *ICSE*, 2014.

[29] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "Conseq: detecting concurrency bugs through sequential errors," in *ACM Sigplan Notices*, vol. 46, no. 3. ACM, 2011, pp. 251–264.

[30] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *ICSE*, 2013.