TFix: Automatic Timeout Bug Fixing in Production Server Systems

Jingzhu He
North Carolina State University
jhe16@ncsu.edu

Ting Dai
North Carolina State University
tdai@ncsu.edu

Xiaohui Gu
North Carolina State University
xgu@ncsu.edu

***Abstract*—Timeout is widely used to handle unexpected failures in distributed systems. However, improper use of timeout schemes can cause serious availability and performance issues, which is often difficult to fix due to lack of diagnostic information. In this paper, we present TFix, an automatic timeout bug fixing system for correcting misused timeout bugs in production systems. TFix adopts a drill-down bug analysis protocol that can narrow down the root cause of a misused timeout bug and producing recommendations for correcting the root cause. TFix first employs a system call frequent episode mining scheme to check whether a timeout bug is caused by a misused timeout variable. TFix then employs application tracing to identify timeout affected functions. Next, TFix uses taint analysis to localize the misused timeout variable. Last, TFix produces recommendations for proper timeout variable values based on the tracing results during normal runs. We have implemented a prototype of TFix and conducted extensive experiments using 13 real world server timeout bugs. Our experimental results show that TFix can correctly localize the misused timeout variables and suggest proper timeout values for fixing those bugs.**

I. Introduction

Timeout is commonly used to handle unexpected failures in complex distributed systems. For example, when a server A sends a request to another server B, A can use the timeout mechanism to avoid endless waiting in case B fails to respond. However, mis-using timeout mechanisms such as using either a too large or a too small timeout value can cause the server system to hang or experience significant performance slowdown [1], [2], [3]. For example, a misused timeout bug caused Amazon DynamoDB to experience a five-hour service outage in 2015 [4]. The root cause of this bug is due to improper timeout value setting under unexpected workload increase. Moreover, timeout bugs are often difficult to fix since most timeout bugs produce no error message or misleading error messages [3]. In this paper, we focus on fixing misused timeout bugs which are caused by misconfigured timeout variables in Java programs. Our previous bug study [3] shows that 47% real world timeout bugs fall into this category.

A. A Motivating Example

To better understand how real-world timeout bugs happen, and how they can affect cloud services, we use the HDFS-4301 [1] bug as one example shown by Figure 1. This

[1]We use "system name-bug #" to denote different bugs.

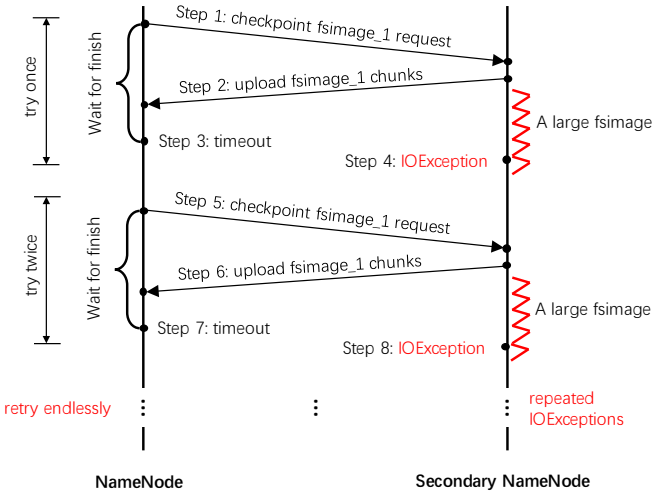

Figure 1: The HDFS-4301 timeout bug. Checkpointing from secondary NameNode to NameNode failed with repeated `IOExceptions`. The root cause of this bug is a too small timeout value configured for transferring large fsimages.

bug is caused by using a too small timeout value (i.e., 60 seconds) for transferring a large fsimage between the primary NameNode and the secondary NameNode. In this bug, the secondary NameNode issues an HTTP GET request to inform the primary NameNode that the latest fsimage is available for checkpointing. The primary NameNode then sends another HTTP GET request to the secondary NameNode to retrieve the fsimage file. When the fsimage file is large and/or the network is heavily congested, it takes the secondary NameNode more than 60 seconds to finish uploading the fsimage, causing the `HTTPURLConnection` timeout between the primary NameNode and the secondary NameNode. The secondary NameNode endlessly repeats the same checkpoint operation because the fsimage uploading keeps failing due to timeout.

Figure 2 shows the major code snippet of the HDFS-4301 bug. The secondary NameNode invokes the `doCheckpoint` function at line #389 in a while loop (line #368-404) to upload the latest fsimage to the primary NameNode, periodically. The primary NameNode sends an HTTP GET request at line #255 to the secondary

Figure 2: The code snippet of the HDFS-4301 Bug. The -·-> represents the function call flows, while the ·····> represents how `IOException` happens and how it is thrown, catched and handled by the Secondary NameNode.

NameNode to retrieve the fsimage. However, the timeout value for the `HTTPURLConnection` is too small, causing the read operation to timeout and throw an `IOException` at line #358. The `IOException` is thrown to the `getFileClient`, `uploadImageFromStorage`, and `doCheckpoint` functions, and finally caught by the `catch` block in the `doWork` function. The secondary NameNode endlessly retries the `doCheckpoint` operation resulting in repeated `IOExceptions`. However, this `IOException` is simply logged at line #390, which does not provide any information about the root cause of the bug, that is, the timeout variable is set to be too small. Even if the developer figures out the root cause is the misconfigured timeout variable, it is still difficult for the developer to come up with a proper timeout value that works for his or her HDFS system.

B. Contribution

In this paper, we present TFix, an automatic misused timeout bug fixing system. TFix leverages TScope [5] to detect a timeout bug in server systems. After a timeout bug is detected, TFix executes a novel *drill-down bug analysis* protocol to automatically narrow down the root cause of the detected timeout bug and produce recommendations for fixing the timeout bug. TFix first determines whether the detected timeout bug is caused by mis-using a timeout scheme. To achieve this goal, we leverage a system call frequent episode mining scheme [6] to check whether any commonly used timeout functions (e.g., `MonitorCounterGroup` function in Flume system [7]) are invoked when the bug is triggered. If the timeout bug is classified as mis-used timeout bug, TFix further localizes the functions that are affected by the timeout bug. Intuitively, when a timeout bug is triggered, the affected function will either run longer time or run more frequently. TFix employs an application performance tracing tool called Dapper [8] to identify timeout bug affected functions. After the function is identified, we use the taint analysis tool [9] to narrow down which timeout variable(s) are used by the affected function. We then perform timeout variable value recommendation based on the profiled execution time of the pinpointed function during normal runs. Specifically, our paper makes the following contributions.

- We describe a novel *drill-down bug analysis* framework which can automatically narrow down the root cause of a misused timeout bug and provide recommendations for fixing the bug.
- We present a dynamic system call analysis scheme that can automatically classify a detected bug as misused timeout bug.
- We describe a hybrid scheme that combines dynamic application performance tracing and static taint analysis to localize the misused timeout variable and provide proper timeout value recommendations for fixing the timeout bug.
- We have implemented a prototype of TFix and conducted extensive evaluation using 13 real world server timeout bugs. Our results show that TFix can correctly classify all tested misused timeout bugs and pinpoint the exact timeout variable that has caused the timeout bug. The timeout values suggested by TFix can effectively correct all the tested misused timeout bugs.

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

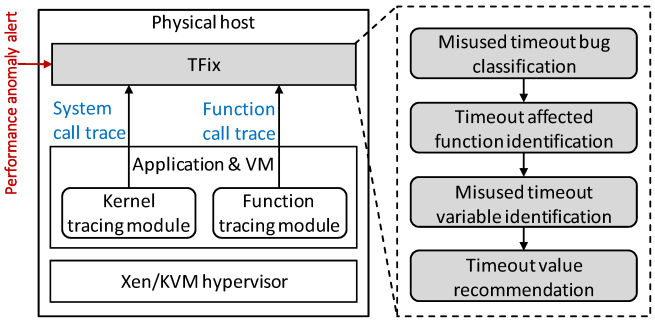

Figure 3: The architecture of TFix.

II. System Design

In this section, we present the design details of the TFix system. We first provide an overview about TFix. We then describe the misused timeout bug classification scheme followed by the timeout affected function identification. Next, we talk about the misused timeout variable identification and timeout value recommendation.

A. Approach Overview

TFix provides a drill-down bug analysis framework for fixing misused timeout bugs, which consists of four major components as shown by Figure 3. When a server system experiences software hang or performance slowdown, TFix leverages TScope [5] to identify whether the anomaly is caused by a timeout bug by analyzing a window of system call trace collected by the kernel tracing module LTTng [10]. If TScope confirms that the performance anomaly is caused by a timeout bug, TFix is triggered to conduct further drill-down analysis. TFix first performs timeout bug classification to determine whether the timeout bug is caused by mis-using certain timeout mechanisms (Section II-B). If the classification result is positive, TFix employs the application function tracing framework Dapper [8] to identify which functions are affected by the misused timeout bug (Section II-C). Next, TFix leverages static taint analysis to localize which timeout variables are used by the identified timeout affected function (Section II-D). Lastly, TFix produces recommendation for the mis-used timeout variable for fixing the timeout bug (Section II-E). The whole drill-down bug diagnosis protocol is executed automatically without requiring any human intervention. We will describe each component in details in the following subsections.

B. Misused Timeout Bug Classification

TFix leverages TScope [5] to determine whether a detected system anomaly is caused by a timeout bug. Timeout bugs can be broadly classified into two groups: 1) *misused timeout bug* where the system anomaly is caused by some incorrectly used timeout variables; and 2) *missing timeout bug* where the system anomaly is caused by lack of timeout mechanisms. In this paper, TFix focuses on fixing misused timeout bug by identifying root cause timeout variables and suggesting proper timeout values for fixing the bug. To achieve this goal, TFix first needs to classify a detected timeout bug as a misused timeout bug. Intuitively, a misused timeout bug is triggered when a certain timeout related function (e.g., `URL.openConnection`, `ServerSocketChannel.open`, `ReentrantLock.tryLock`) is executed. Thus, TFix performs misused timeout bug classification by checking whether timeout related functions are invoked when the bug is triggered.

TFix first provides an offline comparative analysis to extract timeout related functions for each server system. We observe that different server systems often employ different timeout related classes. However, although each system has multiple timeout variables to guard connections, those timeout mechanisms are usually configured by common timeout configuration classes. For example, Flume's timeout mechanisms are built on top of `MonitorCounterGroup` inside the `instrumentation` class, which is used for monitoring system state and building timers. To identify those timeout configuration classes, we employ a dual testing scheme. For each system, we produce a set of test cases each of which consists of two dual parts: one part uses timeout and the other part does not employ timeout. For example, we build a socket connection between the client and HDFS server to write data into HDFS. The difference between the two counterparts is that one has socket write timeout while the other does not. We use HProf [11] to trace the invoked Java functions during the execution of those dual test cases. We compare the lists of the Java functions produced by the two dual test cases in order to extract those functions which only appear in the profiling result of those test cases with timeout mechanisms. To further narrow down the scope of timeout related functions, we only keep those functions that are related to timeout configuration, network connection and synchronization since timeout mechanisms need timers to monitor the elapsed time and timeout mechanisms are often used in network connection and synchronization operations.

After identifying those timeout related functions, TFix needs to employ an efficient scheme to match with those functions during production run. To avoid expensive application function instrumentation, TFix employs a system call frequent episode mining scheme [6] to match with those timeout related functions. The basic idea is to extract unique system call sequences produced by those timeout related functions during the offline analysis. During production run, TFix performs the frequent episode mining over runtime system call sequences and checks whether the frequent system call sequences produced by those timeout related functions exist in the runtime trace. If we find one or more timeout related function matching, TFix classifies the detected bug as a misused timeout bug.

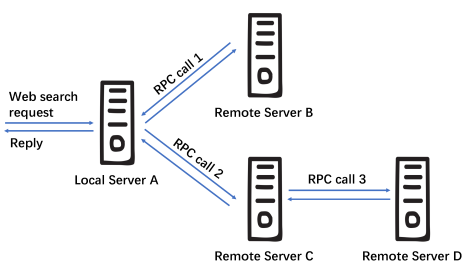

Figure 4: A web search example.

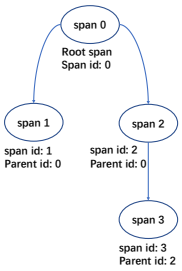

Figure 5: The Dapper trace.

C. Timeout Affected Function Identification

After classifying a detected bug as a misused timeout bug, TFix wants to identify which functions are affected by this misused timeout bug. To achieve this goal, TFix leverages a commonly used application performance tracing tool, i.e., Google's Dapper [8] framework. Dapper allows us to trace the beginning and ending timestamps of all function calls and the control flow graph for the diagnosed bug. We choose Dapper tracing tool because it supports distributed systems and incurs low runtime overhead to production systems. The existing implementations of Dapper tracing can only be applied to RPC related functions. TFix augments the Dapper tracing tool to support all timeout related functions.

Dapper's tracing can be modeled as a tree. The tree nodes are called spans and the edges indicate the control flow between spans. Each span contains a span id and a parent id. The root span does not have a parent id. All spans in the tree share the same trace id. A span represents a RPC connection or a function call, containing the information of both the caller and callee (or the client and server). A span also contains a set of activities including the beginning timestamps, ending timestamps, process name, thread name, and the messages embedded in a RPC or function call.

For example, in Figure 4, a user issues a web search request to the local server A. Server A receives the request and sends the request to the remote server B and C to retrieve the results. Server B stores the data locally and sends the result back to Server A. Server C does not contain the data, thus it sends a request to the remote server D to get the result before responding to Server A. In this case, a simple web search contains four RPC calls between the user and the local server or between servers. When we apply the Dapper framework to trace the control flow of the example in Figure 4, we get a RPC tree, shown in Figure 5. The root span (i.e., Span 0) represents the RPC request and response between the user and Server A. Span 1 indicates the RPC connection between Server A and Server B. Span 2 represents the RPC connection between Server A and Server C. Span 3 illustrates the RPC connection between Server C and Server D. The edge between each span indicates a control flow. For example, Server A receives the user's request and issues RPC call 1 and 2 to Server B and C, respectively. Thus, Span 1 and Span 2 share the same parent span (i.e., Span 0).

```
{"i":"1b1bdfddac521ce8", "s":"df4646ae00070999",
 "b":1543260568612, "e":1543260568654,
 "d":"org.apache.hadoop.hdfs.protocol.ClientProtocol.
     getDatanodeReport",
 "r":"RunJar", "p":["84d19776da97fe78"]
}
```

Figure 6: A trace example of Dapper.

After retrieving a Dapper trace for a target bug, we first extract the execution time and frequency of all the functions invoked when the bug happens. Specifically, we calculate the frequency of each function by simply counting how many times it is invoked in the Dapper trace. We calculate the execution time of each function by subtracting the beginning time from the ending time. Figure 6 shows a Dapper trace example. We can see the Dapper trace is well structured. The trace contains various labels indicating different kinds of information. Among them, "b" and "e" indicate the beginning timestamp and the ending timestamp of a function, respectively. "d" represents the function name and "r" represents the process name.

We further identify the timeout affected functions by checking the abnormality of the functions' execution time and frequency. We need to consider two cases: 1) a timeout value is set to be too large or 2) a timeout value is set to be too small. If the timeout value is set to be too large, the execution time of the timeout affected function is much longer than its execution time during the system's normal run. If the timeout value is set to be too small, the system experiences repeated failures due to frequent timeout. Therefore, the frequency of the timeout affected function is much higher than its frequency during the system's normal run.

For the first case where the timeout value is too large, we identify a function as a timeout affected function by checking whether its execution time is much larger than the maximum execution time during system's normal run. For example, in HBase-13647 and HBase-6684, the timeout value for the RPC connection is misconfigured to be `Integer.MAX_VALUE`. The system works fine under normal state where an HBase client exchanges messages with an HBase server (e.g., HMaster, RegionServer) within tens of seconds successfully. However, when the HBase server fails, the HBase client hangs for about 24 days, causing the execution time of the HBase client's RPC function significantly prolonged. We identify the RPC function as the timeout affected function based on its increased execution time.

For the second case where the timeout value is too small, the system experiences repeated failures due to frequent

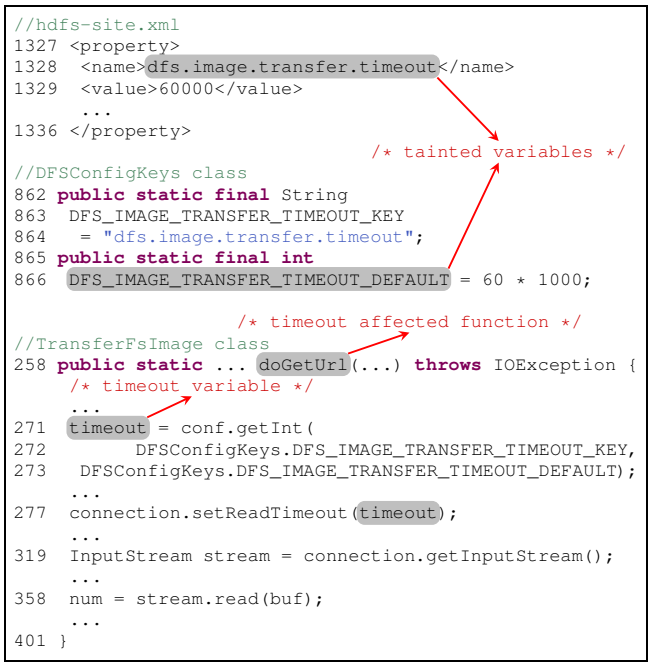

Figure 7: TFix uses the static taint analysis to identify the misused timeout variable for the HDFS-4301 bug.

timeout. Therefore, the frequency of the root cause function greatly increases when the timeout bug is triggered while the execution time of the affected function is similar to the maximum execution time during the system normal run. We thus use frequency to identify those timeout related functions. For example, in the HDFS-4301 bug, the system experiences continuous failures. We identify the `doGetUrl`, `getFileClient`, `uploadImageFromStorage`, and `doCheckpoint` functions as timeout affected functions because of their invocation frequencies significantly increase.

D. Misused Timeout Variable Identification

In this subsection, we describe how TFix identifies the misused timeout variables contributing to the misused timeout bugs. Specifically, we adopt the static taint analysis to correlate the timeout variables with the timeout affected functions to identify the misused timeout variables.

To localize which timeout variable is used when the bug happens, we first retrieve all the timeout variables in the target system. In large scale distributed systems, timeout variables along with other configurable parameters are often stored in specific configuration files [12]. For example, in a Hadoop system, all the configurable variables are defined with default values in configuration files, such as `HConstant` and `DFSConfigKeys` classes. These variables' value can be overridden and customized by users in `.xml` configuration files. Thus, all the variables appear in systems' configuration files and contain "timeout" keyword in their names are potentially related to misused timeout bugs. Next, we taint all these timeout variables and conduct data flow dependency analysis on them to extract all related variables statically. We then check whether the timeout affected functions use the timeout related variables. If a timeout affected function f uses a timeout related variable v_t, we consider v_t as a misused timeout variable candidate. To achieve high accuracy, we also compare the execution time of f with the value of v_t. If they match, we consider v_t as the misused timeout variable.

For example, Figure 7 shows how TFix uses the static taint analysis to identify the misused timeout variable for the HDFS-4301 bug. In this bug, the default timeout value is set to 60 seconds in `DFS_IMAGE_TRANSFER_TIMEOUT_DEFAULT` in `DFSConfigKeys.java`. If users configure the timeout variable `dfs.image.transfer.timeout` in `hdfs-site.xml`, the system uses the configured value. Otherwise, the system uses the default value. We annotate both `dfs.image.transfer.timeout` and `DFS_IMAGE_TRANSFER_TIMEOUT_DEFAULT` as tainted. After applying static taint analysis, we find that the timeout affected function `doGetUrl` uses both tainted variables at line #271-273. Since the user configures the value of `dfs.image.transfer.timeout` in `hdfs-site.xml`, we determine that the misused timeout variable is `dfs.image.transfer.timeout`. We also perform cross validation between the timeout variable value and the execution time of the timeout affected function to confirm whether our timeout variable identification is accurate.

E. Timeout Value Recommendation

After pinpointing the misused timeout variable, TFix recommends a proper timeout value to fix the timeout bug. The timeout value recommendation considers two different cases: 1) the timeout value is too large or 2) the timeout value is too small. As mentioned in Section II-C, if the timeout affected function experiences significant execution time increase, TFix infers that the timeout bug is caused by a too large timeout value. In those cases, TFix recommends to set the timeout value to be the maximum execution time of the affected function right before the bug is detected. Such in-situ profiling results should reflect the system's current environment such as network bandwidth, I/O read/write speed, and CPU load. If the timeout bug is caused by a too small timeout value, we should observe the frequency of the function execution increases. Under those circumstances, TFix suggests a larger timeout value by continuously multiplying the current timeout value by a ratio α, $\alpha > 1$ until the timeout bug is corrected. α is a user configurable parameter which represents the tradeoff between fast fix and larger timeout delay. In our experiments, we set α to be 2.

Table I: 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
HBase	Standalone	Non-relational, distributed database
Flume	Standalone	Log data collection/aggregation /movement service

III. Experimental Results

In this section, we present experimental evaluation results. We have implemented a prototype of TFix and conducted our experiments on a cluster in our research lab. Each host is equipped with a quad-core Xeon 2.53Ghz CPU along with 16GB memory and runs 64-bit Ubuntu 16.04. The system call trace is collected using LTTng v2.0.1. The function call trace is collected using Google's Dapper framework. We first introduce our evaluation methodology. We then present the results of misused timeout bug classification, timeout affected function identification, misused timeout variable identification, timeout value recommendation, and overhead. We also present three case studies to show how TFix correct timeout bugs in details.

A. Methodology

We collected all the bugs from five open source systems. All the systems' names, description and setup mode are listed in Table I. We set up three systems in distributed modes to investigate timeout issues occurring on the communication among different nodes in distributed systems.

We reproduce 13 real-world timeout bugs, including 8 misused timeout bugs and 5 missing timeout bugs. These bugs are collected from bug repositories, e.g., Apache JIRA [13] and Bugzilla [14]. Each report contains detailed information, e.g., version number and system's log information. We list the bugs' description in Table II. In our previous timeout bug identification work [5], the bug benchmarks covered all different root causes presented in the timeout bug study paper [3]. In contrast, TFix focuses on misused timeout bugs. Moreover, TFix only supports Java application systems currently.

We run workloads when the system is in the normal state, in order to approach the real-world system running. The workloads are also listed in Table II. Specifically, for the Hadoop, HDFS and MapReduce systems, we run word count job on a 765MB text file. For the HBase system, we use the YCSB workload generator to make insertion, query and update operations on a table. For the Flume system, we write log events to the log collection tool and distribute the logs repeatedly. These workloads invoke the timeout related functions all of our tested timeout bugs.

B. Results

In this subsection, we first present the classification results for timeout bugs, followed by the identification results for timeout affected function, and then describe the results of localizing the misused timeout variable and timeout value recommendation.

1) Classification results for timeout bugs: TFix classifies a misused timeout bug by checking whether it invokes commonly used timeout functions. As mentioned in Section II-B, our classification scheme matches the runtime system call traces with the frequent system call episodes produced by timeout related functions. Table III shows the classification results. TFix successfully classifies all the 13 timeout bugs. Table III also shows the matched timeout related functions, which are often used for network communications (e.g., `ServerSocketChannel.open`, `URL.<init>`), synchronization operations (e.g., `AtomicReferenceArray.get`, `ReentrantLock.unlock`), and timer settings (e.g., `GregorianCalendar.<init>`, `System.nanoTime`). The results match our assumption that timeout mechanisms are used to protect communications and synchronizations.

2) Timeout affected function identification results: We use the Dapper framework to trace the function calls for tested misused timeout bugs. Dapper has various implementation on different production systems. For example, an implementation of Dapper, HTrace [15] is integrated into Hadoop and HBase. We can configure the parameters for Dapper tracing in the configuration files directly and deploy the production systems to trace the function calls. However, the existing Dapper implementation targets at RPC libraries only. We augment the Dapper implementation by inserting the instrumentation points on synchronization operations and IPC calls. For example, the `setupConnection` function in Hadoop's `ipc.Client` class sets up a connection with IPC server. This `setupConnection` function cannot be traced by the existing HTrace implementation. We formulate the `setupConnection` as a span, which contains all the IPC connection activities, and add annotations to label the function.

Table IV shows the timeout affected functions identified by TFix in all tested misused timeout bugs. For Hadoop-9106, Hadoop-11252(v2.6.4), HDFS-10223, MapReduce-4089, HBase-15645 and HBase-17341 bug, the timeout affected functions have larger execution time compare with that during normal runs. For HDFS-4301 and MapReduce-6263 bug, the timeout affected functions have higher occurrence frequencies with identical execution time during each function run.

3) Localizing the misused timeout variable and timeout value recommendation: We adopt existing static taint tracking framework, i.e., Checker [9], to localize the misused timeout variable. Checker includes various useful plugins

Table II: Timeout bug benchmarks.

Bug ID	System Version	Root Cause	Bug Type	Impact	Workload
Hadoop-9106	v2.0.3-alpha	"ipc.client.connect.timeout" is misconfigured	Misused too large timeout	Slowdown	Word count
Hadoop-11252	v2.6.4	Timeout is misconfigured for the RPC connection	Misused too large timeout	Hang	Word count
HDFS-4301	v2.0.3-alpha	Timeout value on image transfer operation is small	Misused too small timeout	Job failure	Word count
HDFS-10223	v2.8.0	Timeout value on setting up the SASL connection is too large	Misused too large timeout	Slowdown	Word count
MapReduce-6263	v2.7.0	"hard-kill-timeout-ms" is misconfigured	Misused too small timeout	Job failure	Word count
MapReduce-4089	v2.7.0	"mapreduce.task.timeout" is set too large	Misused too large timeout	Slowdown	Word count
HBase-15645	v1.3.0	"hbase.rpc.timeout" is ignored	Misused too large timeout	Hang	YCSB
HBase-17341	v1.3.0	Timeout is misconfigured for terminating replication endpoint	Misused too large timeout	Hang	YCSB
Hadoop-11252	v2.5.0	Timeout is missing for the RPC connection	Missing	Hang	Word count
HDFS-1490	v2.0.2-alpha	Timeout is missing on image transfer between primary NameNode and Secondary NameNode	Missing	Hang	Word count
MapReduce-5066	v2.0.3-alpha	Timeout is missing when JobTracker calls a URL	Missing	Hang	Word count
Flume-1316	v1.1.0	Connect-timeout and request-timeout are missing in AvroSink	Missing	Hang	Writing log events
Flume-1819	v1.3.0	Timeout is missing for reading data	Missing	Slowdown	Writing log events

Table III: TFix's classification result of timeout bugs.

Bug ID	Bug Type	Matched Timeout Related Functions	Correct Timeout Bug Classification?
Hadoop-9106	misused	System.nanoTime, URL.<init>, DecimalFormatSymbols.getInstance, ManagementFactory.getThreadMXBean	Yes
Hadoop-11252 (v2.6.4)	misused	Calendar.<init>, Calendar.getInstance, ServerSocketChannel.open	Yes
HDFS-4301	misused	AtomicReferenceArray.get, ThreadPoolExecutor	Yes
HDFS-10223	misused	GregorianCalendar.<init>, ByteBuffer.allocateDirect	Yes
MapReduce-6263	misused	DecimalFormatSymbols.initialize, ReentrantLock.unlock, AbstractQueuedSynchronizer, ConcurrentHashMap.PutIfAbsent, ByteBuffer.allocate	Yes
MapReduce-4089	misused	charset.CoderResult, AtomicMarkableReference, DateFormatSymbols.initializeData	Yes
HBase-15645	misused	CopyOnWriteArrayList.iterator, URL.<init>, System.nanoTime, AtomicReferenceArray.set, ReentrantLock.unlock, AbstractQueuedSynchronizer, DecimalFormat.format	Yes
HBase-17341	misused	ScheduledThreadPoolExecutor.<init>, DecimalFormatSymbols.initialize, System.nanoTime, ConcurrentHashMap.computeIfAbsent	Yes
Hadoop-11252 (v2.5.0)	missing	None	Yes
HDFS-1490	missing	None	Yes
MapReduce-5066	missing	None	Yes
Flume-1316	missing	None	Yes
Flume-1819	missing	None	Yes

Table IV: The timeout affected functions.

Bug ID	Timeout affected functions
Hadoop-9106	Client.setupConnection()
Hadoop-11252 (v2.6.4)	RPC.getProtocolProxy()
HDFS-4301	TransferImage.doGetUrl()
HDFS-10223	DFSUtilClient.peerFromSocketAndKey()
MapReduce-6263	YARNRunner.killJob()
MapReduce-4089	TaskHeartbeatHandler.PingChecker.run()
HBase-15645	RpcRetryingCaller.callWithRetries()
HBase-17341	ReplicationSource.terminate()

running on the Java compiler. The plugins can check null exceptions, invalidate inputs, tainted variables, etc. We apply the tainted checker on `javac` complier to localize misused variables. Specifically, we select all the timeout variables in configuration files. For each timeout variable, we annotate it as tainted. We compile the system's source code on the `javac` complier. If Checker catches the tainted variable in a timeout affected function, we consider it as the misused timeout variable. Table V shows the misused timeout variables localized by TFix for 8 misused timeout bugs.

Table V also shows the recommended timeout value by TFix. After adopting TFix's recommended value in

Table V: The fixing result of TFix.

Bug ID	Localize the misused timeout variable	Recommended timeout value	Timeout value in the patch	Is bug fixed after applying TFix Recommendation?
Hadoop-9106	ipc.client.connect.timeout	2s	20s	Yes
Hadoop-11252 (v2.6.4)	ipc.client.rpc-timeout.ms	80ms	0ms	Yes
HDFS-4301	dfs.image.transfer.timeout	120s	60s	Yes
HDFS-10223	dfs.client.socket-timeout	10ms	1min	Yes
MapReduce-6263	yarn.app.mapreduce.am.hard-kill-timeout-ms	20s	10s	Yes
MapReduce-4089	mapreduce.task.timeout	100ms	10min	Yes
HBase-15645	hbase.client.operation.timeout	4.05s	20min	Yes
HBase-17341	replication.source.maxretriesmultiplier	27ms	1s	Yes

the system, we find the anomaly does not occur on the system anymore under the same workload. We also list the timeout value in the bugs' patch file in Table V. We observe that the timeout values in the patches are not always correct. When patching misused timeout bugs, developers usually make the timeout variable configurable for users and set the default value to be same as the buggy version before patching. However, it is challenging to make the correct configurations, even for experienced engineers. For example, in the patch of Hadoop-11252(v2.6.4), the default value of the `ipc.client.rpc-timeout.ms` variable is configured to be 0 milliseconds. Developers expose the variable for users to configure. If users do not configure the variable, the timeout bug still happens in a fixed version. As shown in Table V, TFix can fix all the misconfigured timeout bugs. However, TFix's fixing strategy may be different from the patch. We use HDFS-4301 bug as an example. In the patch of HDFS-4301, the default value of `dfs.image.transfer.timeout` is still set to 60 seconds, which is identical with the timeout value before patching. However, the patch limits the chunk size for image transfer, that matched the timeout value. In comparison, TFix changes the timeout value to 120 seconds, that can also fix the problem.

We should note that, the recommended timeout value by TFix might be different under different workloads. This is our design choice, because a fixed timeout setting cannot handle unexpected workload changes or environment fluctuations. For example, in HBase-15645 bug, the misused timeout variable `hbase.client.operation.timeout` defines the time to block a certain table to prevent concurrency issues. Since the table size is small for YCSB workload in our evaluation, the recommended value by TFix is only 4.05 seconds. If we use 20 minutes in the patch under the same YCSB workload, the user will still experience a noticeable delay (about 20 minutes) in the system.

C. Overhead

In this subsection, we discuss the runtime overhead of TFix. TFix's runtime overhead comes from two tracing modules, i.e., system call tracing and function call tracing. Kernel level system call tracing only incurs less than 1% overhead to the system [10]. TFix enables function call tracing (the Dapper tracing) only on a small number of functions which are related to timeout configuration, network connection, and synchronization. We run the workloads on each system with and without tracing. We use the typical benchmarks for all server systems and impose the same types of workloads that trigger the tested timeout bugs. We measure the tracing overhead and list the results in Table VI. We observe that the overhead of TFix in terms of additional CPU load is less than 1%, which makes it practical to apply TFix in real-world production systems.

Table VI: The runtime overhead of TFix.

System	Workload	Average CPU Overhead	Standard Deviation of CPU Overhead
Hadoop	Word count	0.29%	0.023%
HDFS	Word count	0.44%	0.050%
MapReduce	Word count	0.33%	0.012%
HBase	YCSB	0.41%	0.024%

D. Case Study

In this subsection, we discuss three real world bugs in detail to show how TFix works.

HDFS-4301: The root cause and how the bug happens (Figure 1, 2) is already discussed in Section I-A. As mentioned earlier, a misused timeout value (i.e., 60 seconds) cannot keep the `HTTPURLConnection` alive while transferring a large fsimage between the primary NameNode and the secondary NameNode for checkpoint. TFix first successfully classifies the bug as a misused timeout bug, because TFix finds `AtomicReferenceArray.get` and `ThreadPoolExecutor` functions are invoked, when the bug is triggered. As mentioned in Section II-C, TFix then identifies a set of timeout affected functions which have drastically increased invocation frequencies with similar execution times. As mentioned in Section II-D, TFix identifies the misused timeout variable as `dfs.image.transfer.timeout` with the corresponding function `TransferFsImage.doGetUrl()`, using static taint analysis. Last, TFix recommends the timeout value as 120 seconds for the `dfs.image.transfer.timeout` variable by doubling

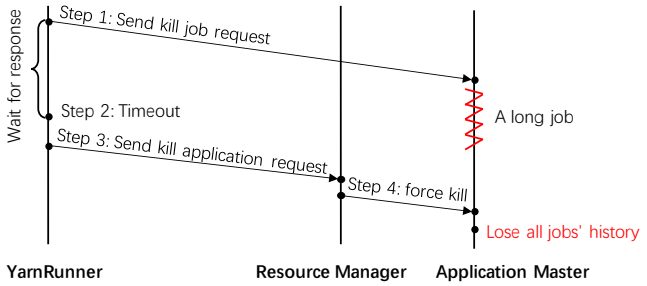

Figure 8: The MapReduce-6263 timeout bug. The ApplicationMaster is forcefully killed, losing all job history after the bug is triggered. The root cause of this bug is the too small timeout value for killing job request sent from YarnRunner to the ApplicationMaster.

the timeout value. We replace 60 seconds with 120 seconds and re-run the workload. We observe the bug does not happen and the NameNodes can successfully finish the checkpoint operation.

Hadoop-9106: This bug is caused by setting too large value to `ipc.client.connect.timeout` variable. The IPC client sets up a connection to the IPC server and the connection timeout value is set to 20 seconds. When the bug happens, the IPC server fails to respond to the IPC client and the IPC client relies on the timeout mechanism to close the connection. Therefore, too large timeout value causes a noticeable delay on the system.

TFix first successfully classifies the bug as a misused timeout bug, because TFix finds the matched timeout related functions `System.nanoTime`, `URL.<init>`, `DecimalFormatSymbols.getInstance` and `ManagementFactory.getThreadMXBean`, when the bug is triggered. TFix then identifies a timeout affected function `Client.setupConnection()` because of its prolonged execution time. Next, TFix pinpoints the misused timeout variable as `ipc.client.connect.timeout` via static taint analysis, because it is used by the `setupConnection()` function. Last, TFix recommends the timeout value as 2 seconds, that is the maximum execution time of `Client.setupConnection()` during system's normal run. We set the `ipc.client.connect.timeout` to 2 seconds and re-run the system. We observed the bug does not happen.

MapReduce-6263: This bug is caused by a too small timeout value for killing MapReduce jobs. As shown by Figure 8, the YarnRunner sends a killing job request to the AppliationMaster with the timeout value set to 10 seconds. However, when the workers are processing a large MapReduce job with limited resources, it takes the ApplicationMaster longer than 10 seconds to finish the job and respond to the YarnRunner. Instead of keeping waiting for the response from the ApplicationMaster, the YarnRunner sends a request to the ResourceManager to kill the ApplicationMaster by force. This force kill results in job history data loss and unavailability of the deployed application.

TFix first successfully classifies the bug as a misused timeout bug, because TFix finds the matched timeout related functions `DecimalFormatSymbols.initialize`, `ReentrantLock.unlock`, `AbstractQueuedSynchronizer`, `ConcurrentHashMap.PutIfAbsent` and `ByteBuffer.allocate`, when the bug is triggered. TFix then identifies a timeout affected function `YARNRunner.killJob()` because of its increased frequency. Next, TFix pinpoints the misused timeout variable as `yarn.app.mapreduce.am.hard-kill-timeout-ms` via static taint analysis. Last, TFix recommends the timeout value as 20 seconds by doubling the current timeout value. We replace 10 seconds with 20 seconds and re-run the system. We observe the bug does not happen and the job finishes successfully.

IV. Limitation

TFix can localize the misused timeout variable if the server system uses timeout variables to in timeout handling operations. However, we observe that some timeout bugs are caused by hard-coded timeout values. For example, in the HBASE-3456 bug, the socket timeout value for HBase client is hard-coded to be 20 seconds in `HBaseClient.java`. To fix the bug, the developer introduces a timeout variable called `ipc.socket.timeout` and make it configurable by the user. We found those kind of timeout bugs mainly occur in early versions of a production system such as Hadoop 0.x version and HBase 0.x version. Although TFix cannot localize misused timeout value under those circumstances, TFix can identify the bug as a misused timeout bug and pinpoint the timeout affected function, which provides important guidance for debugging the problem. As shown in Table II, TFix works well on bugs in Hadoop 2.x and HBase 1.2+ which are more recent stable versions.

In order to provide proper timeout value recommendations, TFix needs to assume that the timeout affected function is invoked before the timeout bug is triggered under the current workload type. However, this assumption does not always hold. Under those cases, TFix cannot provide a proper timeout value recommendation immediately. We can employ prediction-driven timeout tuning scheme to search a proper timeout value iteratively, which is part of our ongoing work.

TFix currently only supports server systems written in Java since we only implement Dapper framework in Java platforms and our implementation of static taint analysis only works on Java files that `javac` compiles. However, our approach is agnostic to programming languages. TFix can be easily extended to support other programming languages by

replacing Java specific Dapper and taint analysis components with other programming language counterparts.

V. Related Work

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

Tracing-based bug detection and diagnosis. Previous work has been done extensively to detect and diagnose bugs using various tracing techniques. For example, X-ray [16] diagnosed performance bugs by tracing the inputs and outputs of different components using dynamic binary instrumentation and inferencing the traces. Chopstix [17] collected low-level OS events including scheduling, CPU utilization, I/O operations, etc. online and reconstructed these events offline for troubleshooting standalone bugs. Fournier et al. [18] proposed to analyze dependencies among processes and how the total elapsed time is distributed using kernel-level tracing. REPT [19] utilized hardware trace to reconstruct the program's execution and employed record-and-replay techniques for debugging. Magpie [20] instrumented middleware and packet transfer points to record fine-grained system events and correlated these events to capture the control-flow and resource consumption of each request for debugging. TScope [5] detected timeout bugs using timeout related feature selection and machine-learning based anomaly detection on system call traces. In contrast, TFix focuses on providing drill down analysis to narrow down the root cause of misused timeout bugs and further provide recommendations for fixing those timeout bugs.

Configuration bug detection and diagnosis. Previous work has been done to study the configuration bugs. Yin et al. [21] and Xu et al. [12] gave comprehensive studies of configuration bugs and pointed out configuration errors are hard to detect and diagnose. SPEX [22] studied configuration constraints and exposed potential configuration errors by injecting errors that violate the constraints. ConfValley [23] introduced a new language to define system validation rules and check configurations against those rules before the application is deployed in production. PCheck [24] analyzed the application source code and automatically emulated the late execution that uses configuration values to detect latent configuration errors. CODE [25] detected configuration bugs by identifying the abnormal program executions using invariant configuration access rules. ConfAid [26] adopted dynamic taint tracking method to instrument the binary application and analyze the information flow in order to pinpoint the root causes of configuration errors. ConfDiagnoser [27] extracted the control flow of configuration options, instrumented the application code for profiling and analyzed the configuration deviation to detect the erroneous configuration option. EnCore [28] applied machine learning techniques to model the correlation between the configuration settings and the executing environment and correlations between configuration entries, in order to learn and detect configuration bugs. However, the existing approaches detect configuration bugs by checking whether the system violates predefined rules. They cannot be readily applied to fix misused timeout bugs which are triggered during system runtime due to unexpected input data or computing environment conditions.

Automatic bug fix. Work has also been done for automatic bug fixes. AFix [29] and CFix [30] proposed automatic patching strategies for concurrency bugs. ClearView [31] identified violated invariants from erroneous executions and generated candidate repair patches to change the invariants. Tian et al. [32] presented an automatic bug fixing patch identification tool to maintain older stable versions. Tufano et al. [33] applied an Encoder-Decoder model based on neural network to mine the existing patches and automatically generate new patches. In comparison, our work focuses on fixing misused timeout bugs with a new drill-down bug analysis approach that can both identify bug root causes and suggest correct timeout values.

VI. Conclusion

In this paper, we have presented TFix, an automatic timeout bug fixing system for production server systems. TFix employs a new drill-down analysis framework for narrowing down the root cause of the misused timeout bug and recommending bug fix. The drill-down analysis of TFix consists of four major steps: 1) checking whether the detected bug is a misused timeout bug by matching common timeout related functions in different server systems; 2) identifying abnormal functions which are affected by the timeout bug using application performance tracing; 3) pinpointing the root cause timeout variable using static taint analysis; and 4) recommending proper timeout values based on the performance tracing results during normal runs. We have implemented a prototype of TFix and evaluated it using 13 real world timeout bugs in a set of commonly used server systems (e.g., Hadoop, HBase, Flume). The experimental results show that TFix can produce effective fix for all the tested misused timeout bugs. TFix is lightweight and imposes less than 1% runtime overhead, which makes it practical for automatically fixing timeout bugs in production systems.

VII. Acknowledgments

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] J. He, T. Dai, and X. Gu, "Tscope: Automatic timeout bug identification for server systems," in *ICAC*, 2018.

[6] 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.

[7] "Flume," https://flume.apache.org.

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

[9] "Checker Framework," https://checkerframework.org.

[10] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[11] "HProf," https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html.

[12] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 70, 2015.

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

[14] "Bugzilla," https://www.bugzilla.org.

[15] "HTrace," http://htrace.incubator.apache.org/.

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

[17] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. L. Peterson, "Lightweight, high-resolution monitoring for troubleshooting production systems." in *OSDI*, 2008, pp. 103–116.

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

[19] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, "Rept: Reverse debugging of failures in deployed software," in *OSDI*, 2018, pp. 17–32.

[20] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *OSDI*, 2004.

[21] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 159–172.

[22] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 244–259.

[23] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou, "Confvalley: a systematic configuration validation framework for cloud services," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 19.

[24] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *OSDI*, 2016.

[25] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. USENIX Association, 2011, pp. 28–28.

[26] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis." in *OSDI*, vol. 10, no. 2010, 2010, pp. 1–14.

[27] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 312–321.

[28] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 687–700, 2014.

[29] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *PLDI*, 2011.

[30] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *OSDI*, 2012.

[31] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 87–102.

[32] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 386–396.

[33] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 832–837.