Chronos: Finding Timeout Bugs in Practical Distributed Systems by Deep-Priority Fuzzing with Transient Delay

Yuanliang Chen*, Fuchen Ma*, Yuanhang Zhou*, Ming Gu*, Qing Liao†, and Yu Jiang*✉
*School of Software, Tsinghua University, KLISS, BNRist, Beijing, China
† Harbin Institute of Technology, Harbin, Heilongjiang, China

***Abstract*—Delays are inevitable in complex distributed environments. Timeout mechanisms are commonly used to handle unexpected failures in distributed systems. However, incorrect timeout handling or implementation errors in timeout mechanisms can lead to system hang-ups or crashes. Such timeout bugs may be crucial and pose a significant threat to the availability and security of distributed systems.**

In this work, we introduce Chronos, a general testing framework for automatically detecting timeout bugs in distributed systems with deep-priority transient delays. First, we propose general runtime delayed libraries that dynamically inject fine-grained delays in a Distributed System Under Test (DSUT). To effectively trigger delays and constantly explore timeout bugs in deep paths, Chronos harnesses a deep-priority guided fuzzing that dynamically generates high-quality delay sequences in the runtime. Then, Chronos utilizes transient delays to eliminate the time overhead caused by actual delays and accelerate the test process. We implemented and evaluated Chronos on four widely used distributed systems, including ZooKeeper, MySQL-Cluster, HDFS, and Go-Ethereum. Compared with the state-of-the-art techniques, Random, Brute-Force, and Coverage-Guided fault injection, Chronos covers 26.40%, 21.69%, and 15.14% more timeout mechanism logic, respectively. Furthermore, Chronos has detected 27 timeout bugs in these real-world applications, which have been repaired by the corresponding maintainers.

1. Introduction

Distributed systems are prone to various faults that may occur at runtime. One of the most common faults is delays, which can be caused by uncontrollable factors (e.g., network traffic, resource monopolization, software bugs, etc.) in complex distributed systems [1], [2]. To mitigate these unexpected failures, various timeout mechanisms have been proposed, ensuring the stability and reliability of distributed systems. For instance, when a distributed node n_1 sends a request to another node n_2, n_1 can use the timeout mechanism to avoid infinite waiting in case n_2 fails to respond. In practice, most timeout mechanisms guarantee one of the most fundamental properties called availability [3], which ensures that distributed systems can consistently provide functional services uninterrupted.

Distributed systems are inherently complex, involving numerous resources and nodes that communicate with each other. To handle unexpected delays and maintain uninterrupted service provision, these systems require a significant number of timeout mechanisms. However, due to this complexity, it can be challenging to avoid incorrect handling or implementation bugs in these timeout logics. Since timeouts play a critical role in the operation of distributed systems, any bugs can have severe consequences. They can result in service unavailability, data loss, and even compromising system security. As an example of real-world distributed system failure, a bug [4] in Facebook's platform is caused by missing timeout checking. As a result, a widespread outage affected the company's own services as well as many third-party sites that used Facebook's authentication services. Another example is ZOOKEEPER-3189 [5], which caused ZooKeeper to stop responding to client requests, leading to a massive outage of service disruption for several hours. Attackers can easily exploit such bugs to conduct distributed denial-of-service (DDoS) attacks, leading to significant financial loss. Such bugs that break the availability of distributed systems by errors in timeout mechanisms are called **Timeout Bugs**.

The timeout mechanisms are challenging to test in reality because such code is infrequently executed. To effectively test these mechanisms, it is necessary to trigger more abnormal delays than those occurring naturally during regular use. Distributed system model checkers [6], [7], [8], [9] enumerate the orders of non-deterministic events (including delays) and suffer from the state space explosion problem. Software fault injection (SFI) is a promising technique to test the resilience and dependability of distributed systems [10], [11]. Chaos engineering [12], [13], [14], [15], [16], such as ChaosBlade [15], is a run-time injection technique that randomly injects various faults into a running software system at the OS level. However, the effectiveness of such coarse-grained delay injection is limited, as it ignores lots of context information, resulting in ineffective testing. Existing compile-time fault injection approaches [17], [18], [19], [20], [21] have shown promising results by injecting faults at the source code level, effectively detecting hard-to-find

*Fuchen Ma has contributed equally to this work.
✉Yu Jiang is the corresponding author.

bugs. Tools [22], [23], [24] utilize bruteforce search to explore the combinations of multiple fault blocks. However, they struggle to effectively explore the vast space of a practical distributed system. To effectively explore the huge state space, FIFUZZ [18] and CrashFuzz [25] employ fuzzing technology to mutate the fault sequence according to the runtime code coverage feedback, successfully detecting numerous bugs in real-world programs. Nonetheless, code coverage-guided algorithms may lead to the exploration of many fault-irrelevant codes, causing inefficiency in exploring fault-handling logic. Additionally, compile-time injection techniques often require manual efforts to identify fault points, limiting their scalability and generality.

To effectively detect timeout bugs in distributed systems, there are three main challenges: (1) **The first challenge** is to provide a general method for precisely injecting fine-grained delays in DSUTs. This can be difficult due to the complex and varied timeout mechanism implementation among nodes in distributed systems. Identifying the appropriate locations to inject delays is crucial for effectively testing these timeout mechanisms. (2) **The second challenge** lies in effectively triggering instrumented delays in deep paths, where timeout bugs tend to remain hidden. In distributed systems, the business logic is usually divided into multiple phases, each with its corresponding timeout mechanism. Frequently executing delays at the shallow execution paths prevents exploring timeout mechanisms in deep paths, resulting in ineffective testing. (3) **The third challenge** is that executing delays will introduce significant overhead to the testing process. Since actual timeouts are usually time-consuming, directly inserting delays reduces testing speed and, consequently, testing performance.

To tackle these challenges, we introduce Chronos, a general testing framework designed to detect timeout bugs in distributed systems. First, Chronos injects the delay logic into the common runtime libraries that are used by the DSUTs. By replacing their original dependent libraries with our delayed libraries in the runtime environment, DSUTs are automatically injected with fine-grained delays. Secondly, to effectively trigger the injected delays and explore timeout bugs hidden in deep paths within DSUTs, Chronos proposes a deep-priority guided fuzzing algorithm. It dynamically mutates the combinations of possible delays, calculates the execution depth of each triggered delay, and prioritizes the delay sequences that can cover deep delay blocks. Finally, to expedite the testing process and eliminate the time overhead caused by actual delays, Chronos introduces the transient delay. It resets the timeout value to zero, immediately triggering the timeout mechanisms in DSUTs without the need to wait for an extended actual time. In this way, Chronos continuously generates plenty of abnormal delays to exercise as many timeout mechanisms as possible and effectively detects timeout bugs in distributed systems.

We implemented Chronos and evaluated its effectiveness on four widely-used distributed systems: ZooKeeper [26], MySQL-Cluster [27], Hadoop Distributed File System (HDFS) [28], and Go-Ethereum [29]. In comparison to other state-of-art fault injection approaches, e.g., Random, BruteForce, and Coverage-Guided, Chronos excelled in exposing more timeout bugs and covering 26.40%, 21.69%, and 15.14% more delay blocks, respectively. Additionally, Chronos identified 27 timeout bugs in total, with 5 in ZooKeeper, 14 in MySQL-Cluster, 6 in HDFS, and 2 in Go-Ethereum.

In summary, we make three key contributions:

- We design and implement the general runtime delayed libraries. Any systems linked to them are automatically injected with fine-grained delays. These common runtime libraries are open-sourced,[1] and can be used directly.
- We introduce a deep-priority guided fuzzing with transient delays to effectively explore timeout mechanisms in DSUTs and detect the timeout bugs in deep paths.
- We implement and evaluate Chronos on four widely used distributed consensus systems. We will open-source Chronos[1] for practical usage. For now, Chronos has successfully detected 27 timeout bugs.

2. Background of Timeout Mechanism

Since a distributed system may encounter different kinds of faults, e.g., delays, the timeout mechanisms are proposed to handle these faults at runtime. Figure 1 shows the main distribution of timeout mechanisms in a typical distributed system. When object O1 sends a request to another object O2, O1 sets a timeout value and waits for the response from O2 until the time expires. In case O2 fails or a message loss occurs, O1 can break out of the waiting state triggered by the timeout event, execute timeout handling code, and take proper actions (e.g., retrying or skipping) accordingly. In general, the timeout mechanism in a distributed system mainly distributes on two parts: (1) Components interact locally through **Local IO**; (2) Nodes communicate remotely via **Network IO** [30], [31], [32], [33], [34].

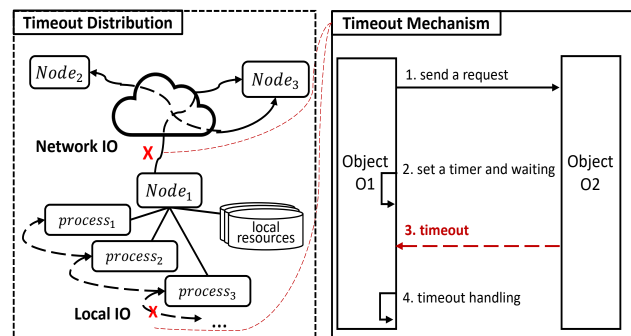

Figure 1. The timeout distribution and timeout mechanism example in a typical distributed system.

Interactions via local IO include intra-process resource access and inter-process communication. Intra-process resource access enables the program to access and manipulate resources, including files, devices, etc., within its own address space [35], [36]. Inter-Process Communication (IPC) is commonly used in the operating system to allow

1. Chronos: https://github.com/SecTechTool/Chronos

processes to communicate and synchronize their actions and resources [37], [38]. To prevent unexceptional faults, e.g., resource monopolization, IO blocking, deadlock, etc., timeout mechanisms are used to prevent a program from becoming unresponsive or hanging indefinitely if a resource/process is unavailable or unresponsive. By setting appropriate timeouts, systems can manage their local interactions more effectively and help improve their performance.

Communications via network IO contain direct communication and indirect communication. In direct communication, nodes interact with each other based on the direct coupling between sender and receiver: the sender has a reference pointer to the receiver and specifies it as an argument of the communication primitive [39], [40]. Indirect communication is defined as communication between entities in a distributed system through an intermediary with no direct coupling between the sender and the receiver(s). The sender does not know the identity of the receiver(s), and vice versa [41], [42]. Timeout mechanisms are essential for managing remote network communication, ensuring the stability and reliability of the system, and preventing issues such as network congestion and unresponsive nodes.

3. Overview

3.1. Definition of Timeout Bugs

Threat Model: Throughout this paper, we use the following threat model. First, we formally define a distributed network as $\phi = \{n, f, T\}$. Specifically, n means the number of normal nodes that perform correctly in the network. f presents the exceptional nodes, which may suffer huge and persistent delays in a hostile runtime environment. T presents the type of fault tolerance mechanism used in the distributed system. The proportion of exceptional nodes should be less than its fault-tolerant threshold. For example, MySQL Group Replication uses the Paxos distributed algorithm [43] to provide distributed coordination between servers. Hence, its fault-tolerant rate is $1/3$, and $f/(f+n)$ should be smaller than $1/3$. We assume that the delay within and between normal nodes is negligible, while the delay within and between exceptional nodes, as well as between exceptional and normal nodes, is dynamic and can be high.

Formally, given a Distributed System Under Test denoted as $DSUT$=$\{P, R, D\}$, where P=$\{p_1, p_2, ..., p_n\}$ represents all processes in the system. R=$\{r_1, r_2, \ldots, r_m\}$ denotes the resources needed, and D=$\{d_1, d_2, \ldots, d_l\}$ represents the set of interactions with timeout mechanisms among P and R, referred to as **delay blocks**. Specifically, a delay block $d_a = [p_i \xrightarrow{t_1} p_j : h_a]$ signifies that process p_i calls p_j with a timeout of t_1, and upon a timeout event, the handler h_a is triggered to deal with it. Similarly, the delay block $d_b = [p_i \xrightarrow{t_2} r_j : h_b]$ represents that p_i requests resource r_j with timeout t_2 and timeout handler h_b. Additionally, we define the symbol $W = \{w_1, w_2, ..., w_k\}$ as the workload for DSUT which contains a set of normal requests.

We define **timeout bugs** as bugs in D that can cause a server crash ($\exists p_i \in P$, where p_i has crashed) or a service hang ($\exists w_j \in W$, where w_j remains unhandled for a long time) in DSUT. We define the testing process as $DelaySeq \xrightarrow{DSUT} State_{error}$. To effectively detect timeout bugs, the main idea is to generate as many delay blocks d_i combinations as possible, which we call $DelaySeq$. For a distributed system under test, after conducting a timeout test that triggers the delay blocks in $DelaySeq$ and executes corresponding timeout handling codes, the system should return to its normal state and provide functional services as usual. Otherwise, the distributed system is in an error state $State_{error}$ and we identify there is a timeout bug detected.

3.2. A Motivation Example

Timeout bugs are hard to detect and can lead to severe consequences in distributed systems. One such example is a timeout bug in the Datastreamer of HDFS (versions 2.7.7 and 3.1.3), where incorrect handling of the timeout mechanism led to critical issues [44]. This bug caused the HDFS service to hang, resulting in data unavailability for all applications using it. Figure 2 illustrates the key steps to trigger this bug, while Figure 3 presents the core code snippet of it. The Datastreamer thread in HDFS manages the stream to datanodes. First, the Datastreamer uses a heartbeat timeout mechanism to wait for data to be stored in the dataQueue. If the request times out, it continually retries to acquire data. Upon receiving the data, the thread proceeds to execute the function ‘createBlockOutputStream()’, which connects to the datanode and establishes a new stream. However, if the connection times out during this process, the Datastreamer throws an IOException and incorrectly sets its state to INTERRUPTED’, as shown in line 18. However, when the thread then performs other actions, such as an RPC (Remote Procedure Call) to NameNode by the function ‘abandonBlock()’, it first checks the current state and fails immediately since its state is ’INTERRUPTED’, triggering this bug. This bug obstructs the RPC to other nodes, causing service hang-ups and affecting the availability of the HDFS. This timeout bug is fixed by clearing the ‘INTERRUPTED’ state, as shown in lines 19-21.

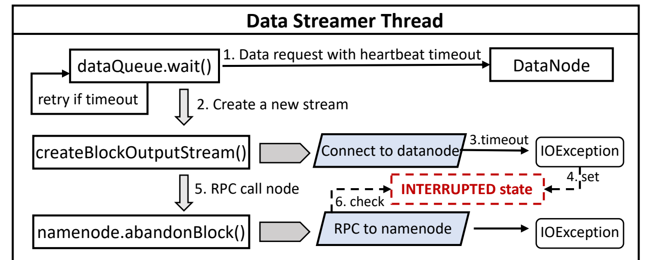

Figure 2. The HDFS-15379 timeout bug blocks and causes the RPC to the namenode to fail, resulting in a service hang for the HDFS system..

Timeout mechanisms are commonly used in distributed systems, and bugs in their implementation are inevitable. Bugs in one node may affect the whole distributed system, thus, causing severe consequences such as service hangs or

```
class DataStreamer extends Daemon {
   public void run() {
      // wait for a packet to be sent.
      while ((dataQueue.isEmpty())) {
         long timeout = 2000; //ms
         try { dataQueue.wait(timeout);
         } catch (InterruptedException e) {
            LOG.debug("ThreadInterrupted", e);
         }
      }
   }
   boolean createBlockOutputStream(...){
      try { s = createSocket(...);
         long t = dfsClient.getimeout(); //5s
         IOStreamPair s = c.socketSend(s,t);
      }catch (IOException ie) {
         errorState.setState(ie)
+        if(ie instanceof INTERRUPTED){
+           //clear thread interrupt state
+           interrupted();}
      }}
}
```

Figure 3. The core code snippet of HDFS-15379. Error timeout handling in function 'createBlockOutPutSteam'. Lines 19-21 are the fixed code.

node crashes. We can draw three important lessons from this case: 1) Timeout mechanisms vary in their implementation in practical scenarios. The complexity and diversity of delay blocks, such as the first delay block d_1 (lines 5-10) and the second delay block d_2 (lines 13-22), make it challenging to accurately identify delay blocks in real-world distributed systems. To handle this challenge, Chronos injects delays at the library layer and identifies delay blocks by analyzing the call trace at runtime. 2) Some timeout bugs are hidden in deep paths, and to trigger them, some prior timeout mechanisms must be bypassed. In this case, a prior timeout mechanism exists within the heartbeat session before the datanode connection process. If this session's timeout is frequently triggered, the subsequent process's timeout will not be triggered, concealing the bug. To address this issue, Chronos employs a deep-priority guided algorithm to dynamically select high-quality delay sequences for exploring as many timeout mechanisms in deep paths as possible. 3) It takes a long time to trigger a timeout mechanism, leading to inefficiencies in one test input execution. In this instance, at least 7 seconds are required to trigger all necessary timeout mechanisms and reveal this bug, significantly slowing down the test speed and impacting the efficiency of bug detection. To solve this problem, Chronos proposes the transient delay mechanism to trigger timeout handling logic immediately.

4. Chronos Design

Design goal: A practical timeout bug detection framework should have the following properties.

- *General:* Chronos is designed to find timeout bugs for most practical distributed systems, from distributed file systems, e.g., HDFS [45], to distributed configuration service, e.g., ZooKeeper [26]. From distributed database systems, e.g., MySQL [46] to de-centralized distributed blockchain, e.g., Ethereum [47]. The tool can be deployed to different distributed systems with minor adjustments.
- *Fine-grained:* To effectively find timeout bugs in distributed systems, the injected delay is designed to be fine-grained to cover as many timeout mechanisms as possible.
- *Accurate:* Chronos is designed to have satisfying precision and recall to avoid reporting false positives.
- *Fast:* Chronos should have a high testing performance.

4.1. Chronos Workflow

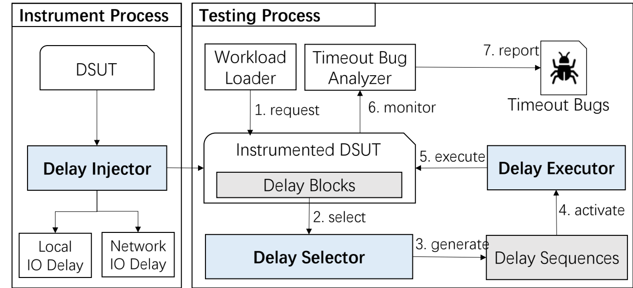

Figure 4. The workflow of Chronos. It includes three main components: (1) Delay Injector for determining where to inject delays. (2) Delay Selector for deciding when to activate delays. (3) Delay Executor for handling how to execute delays effectively.

Figure 4 illustrates the workflow of Chronos, consisting of two main phases. The first phase is the delay instrument process. In this phase, given a DSUT, the Delay Injector first decides where to inject delays, then precisely injects the fine-grained delay logic, and finally outputs the Instrumented DSUT with a set of delay blocks. The second phase is the testing process, which involves the following steps: (1) Chronos first loads workloads to send requests to the distributed system. (2) Then Delay Selector dynamically selects a subset of the delay blocks according to the runtime context (e.g., call trace, execution depth, etc.). (3) Chronos combines the selected delay blocks and generates a delay sequence. (4) Delay Selector activates all delay blocks in the sequence. (5) DSUT executes the workload with the activated delay blocks. To speed up the testing process, the Delay Executor proposes the transient delay mechanism to quickly trigger the timeout mechanisms. (6) Finally, the Timeout Bug Analyzer monitors the runtime states of distributed systems in real time and identifies if there are node crashes or service hangs. (7) The timeout bugs are reported once they are detected. (8) Chronos proceeds to the next iteration (from step 1 to step 7) of the testing process until termination.

4.2. Delay Injector

The Delay Injector determines where to inject delay (which code locations in delay blocks are eligible to inject delays). There are three alternative delay injection methods as shown in Figure 5. (1) Directly inject delays at the source code. (2) Inject delays at the OS level. (3) Inject delays in the runtime library.

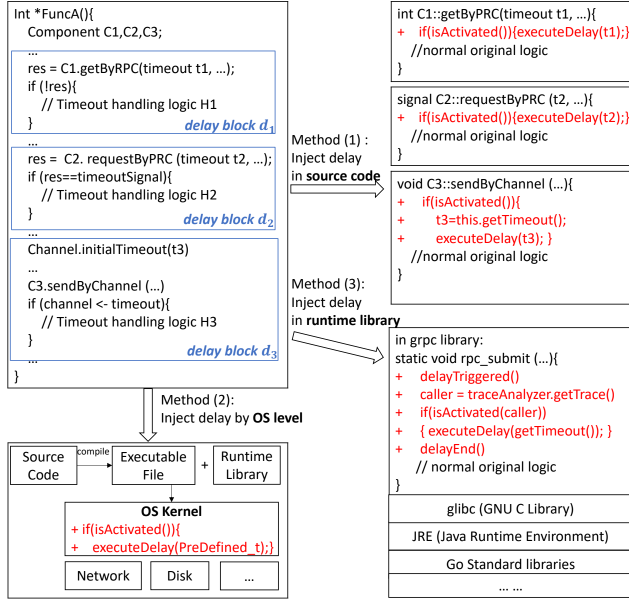

Figure 5. The three alternative delay injection methods. Method (1) injects delays at source code. Method (2) injects delays at the OS level. Method (3) injects delays in the runtime library.

Source Code Injection directly modifies the source code of DSUT. The process begins by identifying all delay blocks where two components interact with the timeout setting. Subsequently, the delay logic is injected into their interactions, as shown in Figure 5. This method allows for fine-grained runtime context analysis and has been widely used by existing compile-time fault injection tools. However, automatically identifying delay blocks in the source code is hard due to the complex and varied implementation of timeout mechanisms across different projects and programming languages. To precisely identify them, we need to search for all calls to the timeout function. There are two types of APIs that contain timeout mechanisms. The first type is Low-level IO-related APIs, e.g. sys.open(), IO.send(), etc., which are few and standardized. The second type is High-level interaction APIs, e.g. rpc_submint(), grpc.send(), netty.write(), etc., which are highly diverse and varied, making it challenging to summarize them manually. Source code usually does not directly call the Low-level APIs, but calls a variety of high-level APIs. Consequently, identifying all API calls to timeout functions in the high-level library requires significant manual efforts, inevitably resulting in missing delay blocks. In addition, source codes may not always be available for testers.

OS kernel Injection in Local IO and Network IO by utilizing the capabilities of the Linux Kernel, such as netem [48]. Netem allows users to inject various network emulation faults, including delay, packet loss, duplication, and corruption, into the network traffic. This method is general and adaptable enough and has been widely used by existing chaos engineering tools. However, such injection is coarse-grained which loses lots of contexts, e.g., call traces. It is hard for the OS kernel to retrieve call traces because we cannot reuse existing userspace or language runtime-specific libraries/tools. Consequently, it can hardly distinguish which caller, delay blocks d_1, d_2, or d_3, calls through the IO. As a result, all delay blocks are activated or deactivated simultaneously, inhibiting the exploration of more delay combination scenarios, e.g., activating d_1 and d_3 while deactivating only d_2. This limitation may hinder the tool's ability to precisely identify and trigger certain timeout bugs in DSUT.

Library Injection injects delay in the runtime libraries of DSUTs. Timeout mechanisms in local interactions and remote communications are executed through local IO and network IO, respectively. Usually, DSUTs do not directly manipulate these IOs but rather utilize existing common runtime libraries. For example, consider the RPC (Remote Procedure Call) in delay blocks d_1 and d_2. All RPCs eventually call the API 'rpc_submit(...)' which in turn calls the 'SocketIO.__send()' in the glibc [49] library. By injecting delay logic into the 'rpc_submit(...)', any RPCs in DSUT are automatically instrumented with the delay logic. In the delay logic, functions 'delayTriggered()' and 'delayEnd()' are designed to record the beginning and end of the delay logic. $if(activated)$ determines whether the delay should be executed and $executeDelay(t)$ actually performs delay for t ms. To distinguish which caller, delay blocks d_1 or d_2, conducts this RPC, Chronos utilizes a traceAnalyzer to trace the call stacks and record the identification of the caller (which delay block), as shown in figure 5. Similarly, for delay block d_3, which eventually calls an API in the Channel library [50], we can inject delay logic into that API as well.

To perform a library injection, Chronos first tracks all IO operations at the runtime library level, e.g., in JRE (Java Runtime Environment), tracking native write APIs in SocketOutputStream for blocking socket messages and tracking write APIs in SocketChannelImpl for non-blocking socket messages, etc. Then Chronos intercepts the relevant APIs in commonly used runtime libraries and injects the fine-grained delay logic into their API implementations. Finally, the runtime delayed libraries are generated. Any program that runs with these libraries will automatically be injected with the fine-grained delay logic.

Library injection achieves both fine-grained delay injection and high generality. Compared with the OS kernel injection, library injection can sense the runtime context of the delay blocks by trace analyzing, allowing for a more fine-grained delay control strategy. Compared with the source code injection, which requires manual effort to identify and inject the delay block for each DSUT, library injection is a one-time effort because the runtime libraries for each programming language are common. The runtime delayed libraries can be reused across multiple DSUTs, making it a practical and general solution.

4.3. Delay Selector

In a DSUT, let D=$\{d_1, d_2, ..., d_l\}$ represents the set of delay blocks, while N=$\{n_1, n_2, \ldots, n_l\}$ denotes the code blocks that are not delay blocks. An executed path in the DSUT consists of a sequence of n_i and d_j blocks. Take

Figure 6 as an example, when all the delay blocks are deactivated and no timeout events occur, then the executed path is $\{n_1, d_1, n_2, d_2, n_3, d_3, n_5\}$. Suppose we activate delay blocks d_1 and d_4, then the executed path would be $\{n_1, d_1, n_7, d_4, n_9, d_5\}$. $DelaySeq$=$[d_i, d_j, ..., d_k]$ represents a combination of delay blocks, where d_i=1 indicates d_i is activated, while d_i=0 means it is deactivated. Our goal is to explore as many d_i and their combinations as possible.

Indeed, the number of delay blocks, l=$sizeof(D)$, in a distributed system can be substantial, leading to a large state space when exploring the delay sequence, where each point can be either activated or deactivated. In total, there are 2^l states. To effectively explore $DelaySeq$, a delay selector is employed to determine when to activate the delay for each $d_i \in D$. In previous state-of-the-art fault injection tools, there are three widely used methods to explore combinations: Random, BruteForce, and Coverage Guided search.

Random Explore: The simplest algorithm, which is adopted by most fault injection tools, randomly selects a subset of delay blocks set D and activates them at random moments. In other words, the statement $if(activated)$ returns true with some probability. However, it ignores the runtime information and cannot explore systematically.

BruteForce Search: Used by FATE [22], PRINCE [23], etc., BruteForce search adopts an enumeration strategy that systematically explores all possible combinations of delay blocks d_i. Specifically, BruteForce first tests all the delay sequences with one delay block, and then tests all the sequences with two delay blocks, and so on. However, it is infeasible when exploring huge state space for practical distributed systems.

Coverage Guided: Similar to FIFUZZ [18] and CrashFuzz [25], the coverage-guided delay selector mutates the combinations of possible delay blocks according to runtime coverage feedbacks and prioritizes the combinations that are prone to increase code coverage. However, code coverage increase does not necessarily equate to delay block increase, resulting in the exploration of many non-timeout logic.

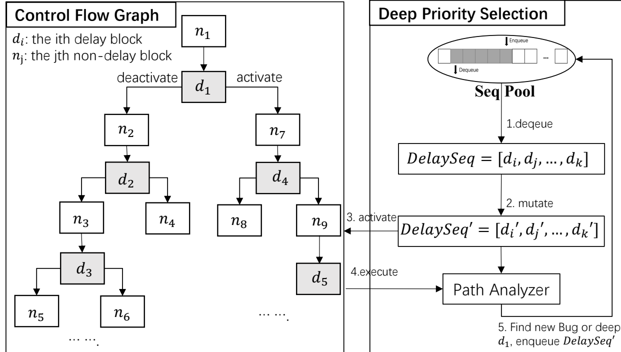

Figure 6. Delay Selector in Chronos. The left part shows the abstract control flow graph of delay blocks in DSUTs. The right part presents the deep-priority selection process.

Deep-Priority Guided. Different from previous work, one heuristic insight of Chronos's delay selector is that delay mechanisms in the deep paths are rarely triggered in normal usage or testing and, therefore, may be more prone to timeout bugs. We use $depth_{d_i}$ to represent the execution depth of d_i, and Avg_D to represent the average depth of the delay blocks in set D. The depth of a delay block is the number of previous delay blocks in its execution path, plus one. We define that $\forall d_i \in D$, if $depth_{d_i} \geq Avg_D$, then d_i is a 'deep delay'; otherwise, it is a 'shallow delay'. For example, considering the path $\{n_1, d_1, n_7, d_4, n_9, d_5\}$ in Figure 6, we have $depth_{d_1}$=1, $depth_{d_4}$=2, $depth_{d_5}$=3, and Avg_D=2. Hence, d_1 is a 'shallow delay', and d_4 and d_5 are 'deep delays'. To help explore deep delays, Chronos utilizes a deep-priority guided algorithm for selecting delay sequences that can trigger newfound deep delays.

Algorithm 1: Deep-Priority Fuzzing Process.

```
   Input  : DSUT: Distributed System under Test
            W: Workloads for DSUT
   Output: T_n: Timeout Bugs
 1 T_n={}  seqPool={};
 2 Detector = setupDetector();
 3 Tracer = setupTracer();
 4 path = DSUT.execute(W, {});
 5 Avg_D, cands = Tracer.triggeredDelays(path);
 6 initSeq = cands.randomInit();
 7 seqPool.enqueue(initSeq) ;
 8 while true do
 9 |   DelaySeq = seqPool.dequeue();
10 |   DelaySeq' = mutate(DelaySeq);
11 |   path' = DSUT.execute(W, DelaySeq');
12 |   for each triggered delay block d_i do
13 |   |   trace_{d_i} = Tracer.getTrace(d_i);
14 |   |   if d_i.isNew() and depth_{d_i} ≥ Avg_D then
15 |   |   |   newSeq = DelaySeq'.append(d_i=1);
16 |   |   |   seqPool.enqueue(newSeq) ;
17 |   |   |   Avg_D, cands.update(d_i);
18 |   |   end
19 |   end
20 |   async:
21 |   |   newBug = Detector.checkBug());
22 |   |   T_n.add(newBug);
23 |   |   seqPool.enqueue(DelaySeq') ;
24 |   end async;
25 end
```

Algorithm 1 illustrates the deep-priority fuzzing process. In the initial phase, as shown in lines 2-7, a bug detector is set up to monitor the runtime status of the system in real time. And a tracer is initialized to help analyze the execution path. DSUT first executes workloads without any delays and generates the execution path. Then, the Tracer extracts delay blocks d_i from the path into a candidate set by analyzing and distinguishing their call traces. At the same time, it calculates the average depth of these delay blocks as Avg_D. Finally, Chronos randomly sets some d_i=1 as the initial delay sequence and puts it into the sequence pool. In each fuzzing iteration, Chronos first dequeues a delay sequence $SeqPool$ and mutates it to $DelaySeq'$ by AFL's bit flips strategy [51] that flips some '0' to '1' and some '1'

to '0' (activate or deactivate some delay blocks). Then, the DSUT executes the same workloads with activated delays in $DelaySeq'$ and naturally triggers some delay blocks. For each triggered d_i, Chronos extracts the call trace as $trace_{d_i}$ and dynamically calculates the depth of d_i. If d_i is newfound and its depth is larger or equal to the average depth, then it is regarded as a new deep delay block. It will be activated and appended into $DelaySeq'$ as a delay sequence. The $newSeq$ will be regarded as an interesting sequence and stored in the sequence pool to guide the subsequent fuzzing process, as shown in lines 15-16. Meanwhile, the bug detector analyzes the runtime status of the DSUT and identifies timeout bugs, as shown in lines 21-23. If any new bugs are found, Chronos records them and enqueues $DelaySeq'$ into the sequence pool. Once all seeds in the queue have been tested for mutation, a fuzz cycle completes. Then the queue resorts all seeds according to their depth. In this way, Chronos constantly generates high-quality delay sequences as test inputs and explores as many deep delay blocks as possible.

4.4. Delay Executor

Transient Delay: In real-world distributed systems, timeouts are typically set between 1 and 3,600 seconds [52]. However, injecting such long delays directly can significantly slow down the testing process. To address this issue, we propose a transient delay mechanism that sets the timeout value to zero. This enables quick triggering of delay logics in the DSUTs and speeds up the testing process.

```
1  // in DSUT source code
2  func getResources(url string) (err error) {
3     client := &http.Client{Timeout: 10}
4     resp, err := client.Get(url)
5     if os.IsTimeout(err) {
6        // A timeout error occurred
7        ...
8        return err
9     }
10 }
11 // in runtime library: go1.20.4/src/net/http
12 func (c *Client) Get(url string) (...) {
13    req, err := NewRequest("GET", url, nil)
14    if err != nil { return nil, err }
15    ...
16    timeout = c.Timeout // original delay=10s
17 +  if(activated) timeout = 0 //transient delay
18    resp, err := send(req, c.ts(), timeout)
19    return resp, err
20 }
```

Figure 7. An example of the timeout mechanism in Go language. In the DSUT source code, it requests a remote URL via the net/http library. The newly added code (line 17) shows the transient delay.

Figure 7 presents a typical example of the timeout mechanism in 'getResource(url)' in the Go language. The function first initializes an HTTP client with a 10-second timeout setting. Then it requests resources by the runtime library 'net/http'. In the HTTP library, it first creates a request, then gets the timeout value, and finally calls native API 'send' of network IO to send the request. The newly added code in line 17 presents the transient delay. When selected by the deep-priority guided delay selector, the transient delay is activated. By directly resetting the timeout value to zero, the timeout mechanism is immediately triggered.

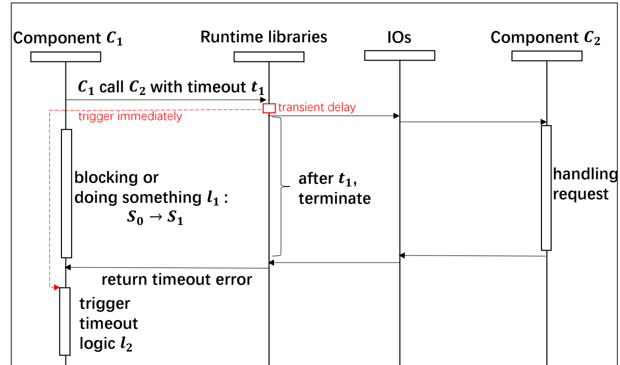

Figure 8. Transient delay in a timeout machenism. In most cases, the transient delay will not introduce false positives.

In a DSUT, the code may make some assumptions about the actual elapsed time and directly setting the timeout value to zero may cause a false positive. Figure 8 shows the core steps of the timeout mechanisms in distributed systems. In the actual delay scenario, when component C_1 calls C_2 with a timeout t_1 via runtime libraries and IOs, C_1 either blocks and waits (synchronous scenario) or does something (asynchronous scenario) that changes the current state of DSUT from S_0 to S_1. Then after time t_1, a timeout error is returned and the timeout handling logic l_2 is executed. If we inject the transient delay in runtime libraries, it immediately returns a timeout error and executes l_2. And the current state of DSUT remains S_0. In most cases, the code l_2 should be independent of code l_1, so there is no false positive, and transient delay is equivalent to actual delay. However in some rare cases, the code l_2 may have some assumptions about code l_1, and its execution logic is determined by the state S_1, then Chronos will output false positives. Section 6.4 will discuss the false positives of Chronos in detail.

To eliminate the false positives introduced by transient delays, Chronos replaces the transient delays with the actual delays (sleep t_1) during the bug reproduction process. If the bug can be reproduced, then it is a true positive. Otherwise, it is a false positive.

4.5. Bug Analyzer

Bug Analyzer is designed to monitor the runtime information of DSUTs and identify their exceptional states, as shown in Figure 9. There are two main types of timeout bugs: server crashes or service hangs.

Server crash: Chronos detects server crash bugs by observing whether the processes or the node of DSUT crashes down throughout the testing procedure. Node Monitor periodically checks whether the process is alive in each distributed node. If any of the processes are down and cannot recover, the Node Monitor reports a timeout bug.

Service hang: Chronos detects service hang bugs by sending normal requests to the DSUTs and monitoring their

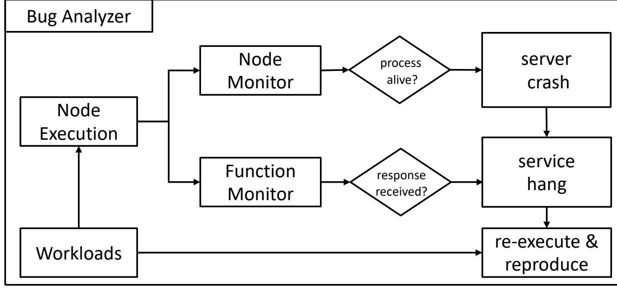

Figure 9. The bug detection process of Chronos. Chronos employs two monitors: Node Monitor for detecting server crashes and Transaction Monitor for identifying service hangs.

responses. The Function Monitor reports a hang timeout bug if it does not receive a response for an extended period. However, the instrumented delays might also cause network traffic, leading to service hangs and resulting in false positives. To mitigate these effects and avoid false positives, the Function Monitor only works during the hang checking phase, after each round of workloads execution, where all instrumented delays are deactivated.

Bug reproduce: Chronos collects all workloads and $DelaySeq$ for each distributed node, sorts them, and stores them based on their executing timestamp. We find the first states that the bug analyzer outputs timeout bugs while processing the workloads and $DelaySeq$, which we call triggering states. We also find the state when a DSUT launches as the starting state. When a timeout bug occurs, Chronos replays these workloads and $DelaySeq$ between the starting state and the triggering state to reproduce the bug and analyze the root cause. To eliminate the false positives introduced by transient delays, all transient delays are replaced by actual delays in the reproduction process.

5. Implementation

We implemented Chronos in the four typical distributed systems: HDFS, ZooKeeper, MySQL, and Go-Ethereum. The reasons we choose them are listed below:

System Popularity: HDFS [45] is a vital component of the Apache Hadoop project [53], which is one of the most widely used data storage file systems in the world. MySQL-Cluster [27] is a highly scalable, real-time, ACID-compliant transactional database. The low TCO (total cost of ownership) and multi-master distributed architecture make it widely used in a variety of applications [54]. ZooKeeper is one of the most popular distributed process coordination. It allows distributed processes to coordinate with each other through a shared hierarchical namespace with high throughput and low latency. Ethereum is one of the most widely used public blockchains (de-centralized distributed systems) in the world with the highest market cap $211.61B [55].

Platform Diversity: These distributed systems come from different organizations with implemented languages. HDFS and ZooKeeper are developed by Apache Software Foundation in Java language. MySQL-Cluster is developed by MySQL AB in C++ language. Go-Ethereum is developed by Ethereum Org in the Go language. Implementation and evaluation of these distributed systems can demonstrate that Chronos is a cross-platform and language-independent testing framework with high generality.

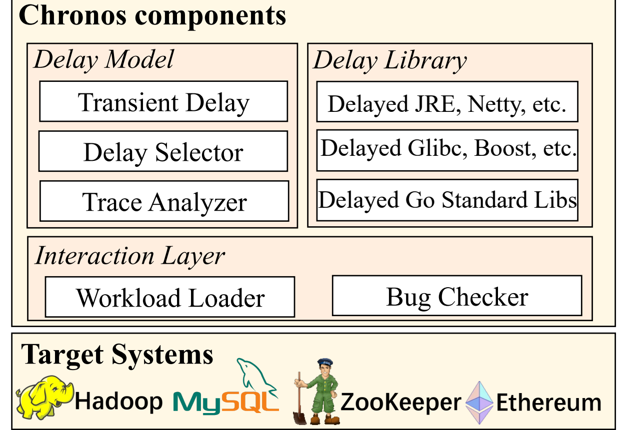

Figure 10. Components of Chronos are divided into three parts – Interaction Layer, Delay Model, and Delay Library.

Figure 10 presents the components of Chronos, which can be divided into three main parts. The first part is the Interaction Layer, which is designed to interact with target systems, including sending workloads to DSUTs and monitoring bugs within them. The second part is the Delay Model which is implemented for generating high-quality transient delay sequences to explore deep logic in DSUTs based on their runtime traces. The third part is the Delay Library which contains runtime libraries instrumented with deep-priority transient delays. The rest of the section describes notable implementation details.

Workload Loader: For Hadoop Distributed File Systems, the workload is collected from Intel HiBench [56], a widely used testing workload suite. For MySQL-Cluster, the workload is generated by SQLancer [57], one of the widely used SQL generators for testing database systems. For ZooKeeper, we developed an automatic tool to generate the workload based on its API documentation and instruction manual. The tool is also open-source and available in our git[1]. For Go-Etheruem, the workload is generated by chainhammer [58], which is an art-of-the-state tool for generating test transactions for blockchain systems.

Trace Analyzer: In the deep-priority guided delay selector, before calculating the depth of each delay block d_i, Chronos first obtains the call trace and identifies the caller (which delay block). In Java language, the trace analyzer is implemented by 'Thread.currentThread().getStackTrace()'. In C++, we use 'execinfo.backtrace()' and 'backtrace_symbols()' to implement the trace analyzer. In Go, the trace analyzer is implemented using the functions 'debug.stack()' and 'CallersFrames()'. The $depth_{d_i}$ can then be calculated by counting the previous $\langle delayTriggered(), delayEnd() \rangle$ pairs in its execution trace.

TABLE 1. 27 TIMEOUT BUGS DETECTED BY THE TOOLS WITHIN 24 HOURS. CHRONOS FOUND ALL 27 TIMEOUT BUGS, INCLUDING 5 IN ZOOKEEPER, 14 IN MYSQL-CLUSTER, 6 IN HDFS, AND 2 IN GO-ETHEREUM. $Random$ FOUND 5 BUGS, AND $BruteForce$ FOUND 6 BUGS, AND $CoverageGuided$ FOUND 9 BUGS RESPECTIVELY.

#	Platform	Bug Type	The Root Cause Analysis	Identifier
1	ZooKeeper	Hang	DeadLock in ZooKeeper node when both sendThread and reconnection timeout and tries to update states.	ZooKeeper-2023481
2	ZooKeeper	Crash	SIGSEGV in NIOServerCnxnFactory when follower nodes create plenty of reconnections after timeouts.	ZooKeeper-2023570
3	ZooKeeper	Hang	Endless EOF Exception in "loadDataBase" after resource request timeouts, leading to service blocking.	ZooKeeper-2023482
4	ZooKeeper	Crash	The leader node breaks down when trying reconnection in SocketNIO after constant timeout exceptions.	ZooKeeper-2023571
5	ZooKeeper	Hang	The following node repeatedly throws EndOfStreamException, stop reading data from the master node.	ZooKeeper-2023480
6	MySQL-Cluster	Crash	SEGV cased by nullptr after a series of reconnection timeouts in MySQL-Cluster breaks down multiple nodes.	MySQL-S1787454
7	MySQL-Cluster	Crash	SEGV in sql_executor caused by error pointer manipulating in timeout handling crashes MySQL server node.	MySQL-S1787465
8	MySQL-Cluster	Hang	SQL selection process is blocked after handling various communication timeouts in the network.	MySQL-S1785100
9	MySQL-Cluster	Crash	Server node crashes down when connection timeout during the data syncing and validation process.	MySQL-S1787522
10	MySQL-Cluster	Crash	Heap-buffer-overflow occurs in "Item_read" after recovering from the direct communication timeouts.	MySQL-S1787551
11	MySQL-Cluster	Crash	SEVG in SQL planer after triggering a series of timeout mechanisms when accessing the local database.	MySQL-S1785171
12	MySQL-Cluster	Crash	Heap-buffer-overflow happens when dealing with various timeout SQL results in the union process.	MySQL-S1787533
13	MySQL-Cluster	Hang	Frequent timeouts in the transaction syncing process hangs the server which stops the SQL execution.	MySQL-S1719125
14	MySQL-Cluster	Hang	Constant timeout requests from other nodes keep blocking SQL optimizer and stop handling SQL queries.	MySQL-S1787477
15	MySQL-Cluster	Crash	SEGV in SQL optimizer when handling timeout SQL queries after multiple reconnections from the network.	MySQL-S1719139
16	MySQL-Cluster	Crash	Repeated timeout error in "item_subselect" crash down the MySQL server node and cannot be recovered.	MySQL-S1734011
17	MySQL-Cluster	Crash	SEGV occurs after packet timeouts when the master node dispatches commands to other slave nodes.	MySQL-S1787473
18	MySQL-Cluster	Crash	SEGV in data syncing process after multiple request timeouts and retry, breaking down the server nodes.	MySQL-S1732431
19	MySQL-Cluster	Hang	Memory leak in "buf_block_init", hanging the server after plenty of SQL requests timeouts.	MySQL-S1787458
20	HDFS	Hang	The AsyncDispather thread is interrupted due to timeout connections and the RM cannot be contacted.	HDFS-20230655
21	HDFS	Hang	The IPC call is interrupted and the RM no long gives any response to the datanode after multiple timeouts.	HDFS-20231630
22	HDFS	Hang	Yarn gets stuck and repeated throw exceptions during the runtime due to timeouts in sending packets to RM.	HDFS-20231643
23	HDFS	Hang	Namenode tries to delete a container that has already been deleted in cleanup process after request timeouts.	HDFS-20231621
24	HDFS	Hang	A deadlock occurs and the MapReduce is rejected to execute when datanode throws a timeout exception.	HDFS-20231134
25	HDFS	Hang	Service hang when Socket is reset and the executing task cannot finish in time after multiple timeouts.	HDFS-20231011
26	Go-Ethereum	Crash	Serial of timeout messages cause panic in the "Full sync process ", and break down Ethereum nodes.	GETH-10365
27	Go-Ethereum	Hang	Repeated timeouts in BlockHeader syncing hang the message queue and stop transaction handling process.	GETH-10363

Adaption to New Distributed Systems: According to a survey of languages used in distributed systems, the vast majority of the distributed systems are implemented in Java, C++, and Go languages [59], [60], [61]. Hence, for most distributed systems implemented in these languages, Chronos can be directly used to detect timeout bugs without modifications, thanks to its general design and the common runtime delayed libraries. Only the Workload Loader needs to be changed accordingly. For DSUTs implemented in other languages, the Chronos framework is also scalable enough to adapt to them. Testers only need to follow three steps to generate the runtime delayed library in a new language: (1) First identifying IO-related basic APIs in the target language, e.g. SocketOutputStream.send(), SocketChannelImpl.read(), etc. in Java. This step needs manual effort and expertise. (2) Statically scan and track all API usage in the runtime library to identify the delay blocks. (3) Insert transient delay logic in the delay blocks before calling those APIs. Steps (2) and (3) are automatic. Furthermore, instrumenting delays in the runtime libraries is a one-time effort for each language. Therefore, the effort of adapting Chronos to other distributed systems implemented in a new language is acceptable.

6. Evaluation

To evaluate the effectiveness of Chronos, we compared it with three state-of-the-art fault injection methods that have been widely used in previous work: Random, BruteForce, and Coverage-Guided Fuzzing on four widely used distributed systems. We ran each distributed system in a cluster of 20 virtual nodes isolated by Docker [62]. Each Docker has a 2.25 GHz 6-core CPU, 16 GB of RAM, and a 480 GB SATA SSD. They all connect to each other with a 10 Gbps network bandwidth setup. They ran Ubuntu 20.04.2 with Linux kernel version 4.4.0. All Docker containers run in a physical machine, which is a 64-bit machine with 128 CPU cores (AMD EPYC 7742 64-Core Processor), and 512 GB main memory. All the experiments are conducted several times with the same workloads, and the average values are used in this paper. We designed experiments to address the following research questions:

- **RQ1:** Is Chronos effective in finding timeout bugs of real-world distributed systems?
- **RQ2:** Can Chronos cover more timeout mechanism logic in distributed systems as compared to other state-of-the-art approaches?
- **RQ3:** Does the transient delay executor effectively improve testing performance?
- **RQ4:** What is the accuracy of Chronos?

6.1. Timeout bugs in real-world distributed systems

We applied Chronos to all four DSUTs for timeout bug detection evaluation. BruteForce fault injection tools, e.g., FATE [22], and Coverage-Guided fault injection tools, e.g., CrashFuzz [25] and FIFUZZ, do not support delay injection. They cannot detect timeout bugs as they do not recognize delay blocks and do not insert transient delays. To make a fair comparison, we have two options. (1) We adapted Chronos to them. First, we manually replaced their fault points with our delay blocks. Then we changed their fault injection to our transient delay injection. However, it requires finding all the delay blocks in the source code first, which is hard and will often result in missing some delay blocks. (2)We implemented their algorithms into Chronos. First,

we instrument code coverage collection logic into DUSTs, then we replace the deep priority guidance with existing algorithms in the fuzzing process. Specifically, we adapted the BruteForce algorithm of FATE to Chronos, which we call $BruteForce$. We adapted the coverage-guided algorithm of CrashFuzz to Chronos, which we call $CoverageGuided$. We ran all the tools on the same distributed systems using the same experimental setup. Chaos testing tools, e.g., ChaosBlade [15], randomly inject different kinds of faults, such as IO delay, CPU burn, memory burn, etc., into a running system. For a fair comparison and to focus on timeout bugs, we adapted the random algorithm of ChaosBlade to Chronos and replaced its actual delays with transient delays, which we call $Random$. Each experiment is conducted for 24 hours. In total, Chronos found 27 timeout bugs on four target distributed systems with 5 in ZooKeeper, 14 in MySQL-Cluster, 6 in HDFS, and 2 in Go-Ethereum. The detailed information on these previously unknown timeout bugs is presented in Table 1.

All 27 found timeout bugs have been confirmed and fixed by the corresponding vendors. Among these, 13 (48.1%) timeout bugs caused server nodes to crash, and the distributed nodes could not recover automatically. The remaining 14 (51.9%) caused the distributed services to hang, making them unable to recover themselves. Some of the timeout bugs can lead to serious consequences. Take bugs #26 and #27 for example, attackers can crash or hang some target nodes and stop their transaction handling or block mining process by producing certain delay strategies, which may directly cause financial loss of ETH (Ethereum cryptocurrency). MySQL-Cluster had the highest number of bugs, with 14 bugs (more than half) being detected. One reason why C/C++ applications might have more bugs is that they tend to be more prone to memory-related bugs, such as buffer overflow (bug #10 and #12), memory leak (bug #17), and misused pointers (bugs #6, #11, and #18). These detected bugs remind us that in addition to the correctness of the timeout handling logic, we also need to be cautious about memory-related manipulation in code implementation.

TABLE 2. BUGS FOUND BY CHRONOS AND OTHER STATE-OF-THE-ART METHODS. OTHER METHODS DETECT NO MORE THAN 7 BUGS, WHILE CHRONOS DETECTS 27 UNKNOWN TIMEOUT BUGS.

Method Name	Number	Bugs ID #
Chronos	27	#1 - 27
$Random$	5	#3, 8, 13, 16, 21
$BruteForce$	6	#3, 8, 9, 10, 16, 21
$CoverageGuided$	9	#3, 7, 8, 10, 13, 16, 19, 21, 26

Comparison with existing methods: In our 24-hour experiments, Random only found 5 bugs, including bugs #3, #8, #13, #16, and #21. BruteForce exploring only found 6 bugs, including bugs #3, #8-10, #16, and #21. Coverage-Guided algorithm has successfully found 9 bugs (#3, #7, #8, #10, #13, #16, #19, #21, #26). However, the rest of the 17 bugs were not found by them because these timeout bugs are hidden in the deep path. To trigger them, many processes with different timeout interactions in multiple phases should be conducted first. With the help of the deep-priority guided transient delays, Chronos successfully explored the delay blocks hidden in deep paths and detected all 27 timeout bugs, proving the effectiveness of Chronos in detecting timeout bugs in real-world distributed systems, which adequately answers **RQ1**. Compared with other state-of-the-art fault detection techniques, Chronos found all the bugs that other methods found.

6.1.1. Case Study. Now we use two cases to illustrate how the timeout bugs detected by Chronos affect the whole distributed system, and how Chronos detects them. **The first case** is the bug #1 listed in Table 1. This is a service hang bug where a deadlock occurs in ‘packetqueue’ caused by incorrect handling of timeout events. This timeout bug can cause a large area of zookeeper service downtime and the developer has already fixed it. It is found in version 3.4.14 of ZooKeeper. The code snippet in Figure 11 describes the details of this timeout bug.

```
1  class SendThread extends ZooKeeperThread {
2     public void run() {
3        if (timeRwServer >= RwTimeout)
4           cleanup();
5     }
6     private void cleanup() {
7        synchronized (packetQueue)
8           for (Packet p : packetQueue)
9              conLossPacket(p); //needs
                     ConnState lock
10    }
11 }
12 public ReplyHeader reconnect(...)(
13    synchronized (ConnState) {
14       if (r.getErr() ==
              Code.REQUESTTIMEOUT.intValue())
15 -           sendThread.cleanup()
16 +           sendThread.interrupt()
17    }
18 }
```

Figure 11. A timeout bug that causes deadlock in the packet handling process and hangs the ZooKeeper system.

Root Cause: When a ZooKeeper node connects to the network, it sets up a SendThread for sending requests to other server nodes. If a request times out, the ‘cleanup’ function is called, which acquires the packetQueue lock. In the ‘reconnect’ function, it first applies the ConnState lock to check the state and then finishes processing the packet. However, if the main thread attempts to reconnect to the server and the reconnection times out, it will acquire the ConnState lock, and then call ‘SendThread.cleanup()’. This will require waiting for the packetQueue lock. The deadlock occurs when the main thread acquires the ConnState lock and waits for the packetQueue lock, and the SendThread acquires the packetQueue lock but waits for the ConnState lock. The developer has fixed this timeout bug by avoiding a direct call to ‘cleanup’. Instead, they interrupt the sendThread and let it clean up its own state (lines 15-16).

This bug is difficult to detect because, in most cases, before the main thread executes the timeout handling code in lines 16-18, the SendThread has already finished the ‘cleanup’ process. The timeout for reconnection is typically set to 5 seconds, and SendThread can usually clean up all

packets within that time and release the ‘packetQueue’ lock. The bug can only be triggered if the queue length is long enough that SendThread cannot release the lock in time, which is rare, and if there happens to be a network delay at that moment. And these two delay blocks are hidden in the deep path, to trigger them, at least five previous delay blocks should be deactivated and two previous delays should be activated. Fortunately, with the fine-grained instrumented delays and deep-priority transient delay, Chronos quickly triggers the timeout mechanisms in both the SendThread and reconnecting process, allowing it to detect this timeout bug in the deep path.

The second case is the bug #6 listed in Table 1. This bug is a severe server crash and attackers may utilize it to cause arbitrary distributed MySQL nodes to break down by conducting a certain delay strategy. It is caused by an implementation bug that incorrectly manipulates the memory pointer. It is found in version 8.0.31 of MySQL-Cluster. The code snippet in Figure 12 describes the detailed information.

```
1  bool check_and_report(THD *thd, int error) {
2    ...
3    if (error==TIMEOUT){...} //retry if timeout
4    else{          //output the error info else
5  +     Diagnostics_area *da= thd->get_stmt_da();
6  -     if (!thd->get_stmt_da()->is_error() &&
7  -         has_temporary_error(the,
       da->is_error())
8  +     if (da!= nullptr && !da->is_error() ||
9  +         has_temporary_error(thd,
       da->is_error())
10    }
11 }
12 bool Slave_worker::reconnect(THD *thd, ...){
13   ...
14   if (!check_connect && _timeout())
15       thd->cleanup(); //clean stmt_da to nullptr
16 }
```

Figure 12. A timeout bug that crashes MySQL nodes.

Root Cause: The Function ‘check_and_report()’ is designed to output detailed error statements by the ‘*da’ pointer. However, it can be cleaned to a null pointer, leading to a SEGV in MySQL nodes if the connection is constantly lost and the reconnection is timing out, as shown in lines 12-16. This timeout bug has already been fixed by adding the nullptr checking, as shown in lines 8-9.

This bug is hard to detect because, in most cases, line 15 is executed due to poor network conditions. Under such circumstances, the ‘check_and_report’ function executes line 3 instead of the else logic (lines 4-10) unless the network happens to be recovered at that moment, which is rare and prevents the bug from manifesting. However, based on fine-grained delay instrumentation and the deep-priority transient delays, Chronos has a high probability of activating the deep delay in ‘reconnect’ while simultaneously deactivating the shallow delay in ‘check_and_report’, effectively detecting this timeout bug hidden in the deep path (there are six previous delay blocks, with 2 activated and 4 deactivated).

6.2. Effectiveness on Timeout Logic Coverage

To evaluate the capacity of Chronos in timeout logic coverage of distributed systems, we set up a network for each target system and compared Chronos with other state-of-the-art methods in the same experimental setup. The delay block (as defined in section 3.1, delay block d_i represents timeout mechanism in DSUT) coverage for each tool in 24 hours was collected. The statistics are shown in Table 3. In conclusion, Chronos always outperforms other methods on all four distributed systems. Compared to methods $Random$, $BruteForce$, and $CoverageGuided$, Chronos covers 26.40%, 21.69%, and 15.14% more delay blocks on average. The statistics adequately answer **RQ2**.

TABLE 3. DELAY BLOCK COVERAGE ON FOUR DSUTS IN 24 HOURS. CHRONOS COVERS 26.40%, 21.69%, AND 15.14% MORE DELAY BLOCKS COMPARED WITH OTHER METHODS.

	Random	BruteForce	Coverage-Guided	Chronos
HDFS	2947	2915	3178	3620
MySQL-Cluster	1651	1692	1733	2116
ZooKeeper	867	937	944	1032
Go-Ethereum	939	1019	1105	1273

Compared with $Random$, Chronos covers 22.84%, 28.16%, 19.03%, and 35.56% more delay blocks on HDFS, MySQL-Cluster, ZooKeeper, and Go-Ethereum, respectively. The reason is that Chronos utilizes finer-grained delays and dynamically adjusts the delay sequence accordingly in the runtime. While $Random$ only performs delay injection randomly, without utilizing runtime information of target nodes. Compared with $BruteForce$, Chronos always outperforms it on all four distributed systems. On average, Chronos covers 21.69% more delay blocks. It demonstrates that deep-priority guided search performs better than brute-force search when exploring delay combinations in DSUTs. Compared with $CoverageGuided$, Chronos also achieves 15.14% more delay block coverage on average in these four distributed systems. The main reason is that the sequence mutating strategy in $CoverageGuided$ relies on code coverage feedback, which does not necessarily reflect the coverage of timeout mechanism codes. As a result, it becomes inefficient in exploring delay blocks.

To observe the trends of coverage growth over time, we record the delay block coverage every minute over 24 hours, as shown in figure 13. According to the figure, Chronos's delay block coverage grows significantly in the first eight hours on all four target distributed systems. After around 20 hours, the coverage of Chronos gradually converges (only less than 1% coverage improvement is observed). As for $Random$, $BruteForce$, and $CoverageGuided$, the coverage grows rapidly in the first 720 minutes. After that, the delays they activated can hardly cover more delay blocks than it does at the beginning of the testing process. Due to the transient delay mechanism, the coverage of all tools converges within 24 hours. Chronos consistently outperforms $Random$, $BruteForce$, and $CoverageGuided$ in terms of delay block coverage across all four distributed systems,

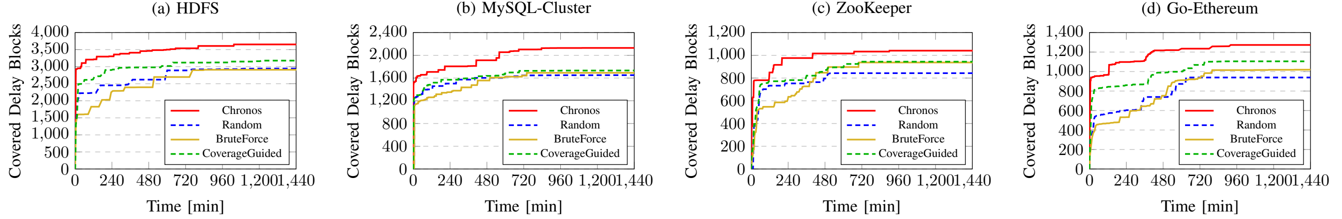

Figure 13. Coverage trends evaluated for Chronos, $Random$, $BruteForce$,and $CoverageGuided$. Compared with them, Chronos with the deep-priority guided algorithm shows better delay block coverage all the time on all the target distributed systems.

thanks to its deep-priority guided sequence selection for exploring more delay blocks.

TABLE 4. CODE COVERAGE ON FOUR DSUTS IN 24 HOURS. CHRONOS STILL OUTPERFORMS OTHER METHODS.

	Random	BruteForce	Coverage-Guided	Chronos
HDFS	28302	28194	31497	32402
MySQL-Cluster	39790	39984	44534	47595
ZooKeeper	8584	9202	10344	10502
Go-Ethereum	10083	10901	11532	12607

To better understand the delay block coverage and the code coverage (widely used in previous works), we also collected code coverage for each tool in 24 hours on these distributed systems. The results are shown in table 4. Compared with $Random$ and $BruteForce$, Chronos achieves 20.36% and 15.93% more code coverage. Even compared with $CoverageGuided$, Chronos still outperforms it on all four DSUTs and covers 5.15% more code on average. It demonstrates that the deep-priority algorithm successfully explores more timeout mechanism logic, which in turn contributes to the code coverage increase.

To explore the main reason why Chronos performs better, we further manually analyze the coverage difference between the coverage-guided algorithm and Chronos. We find that most of the code covered by the coverage-guided method is non-timeout logic. While coverage-guided fuzzing helps cover more code at the beginning of testing, a substantial part of this code corresponds to non-delay blocks. Consequently, many delay sequences, which cannot contribute to new delay blocks, are regarded as seeds. Subsequent mutated delay sequences generated based on them become inefficient or even redundant. As the inputs are in the form of delay combinations, the failure to discover new delay blocks obstructs the generation of new delay combinations, causing the coverage to converge early. On the contrary, Chronos constantly explores new deep delay blocks, which in turn contributes to richer delay combinations. So Chronos performs better on both delay block and code coverage.

6.3. Effectiveness of Transient Delay

To evaluate the effectiveness of the Transient Delay, we also conducted the experiment that compares Chronos with $Chronos^-$, a version of Chronos that disables the transient delay and executes the actual delay (sleep(t)) instead. We collected the delay block coverage and the number of bugs in 24 hours on all four target distributed systems.

TABLE 5. COMPARISON OF $Chronos^-$ AND CHRONOS ON FOUR DSUTS IN 24 HOURS. CHRONOS WITH TRANSIENT DETECTS 92.86% MORE BUGS AND COVERS 18.10% MORE DELAY BLOCKS.

	Number of Bugs		Delay Block Coverage	
	$Chronos^-$	Chronos	$Chronos^-$	Chronos
HDFS	3	6	3095	3620
MySQL-Cluster	8	14	1663	2116
ZooKeeper	2	5	858	1032
Go-Ethereum	1	2	1023	1273
Improvement	-	+92.86%	-	+18.10%

As shown in table 5, with the help of the transient delay, Chronos can detect all 27 bugs in 24 hours, while $Chronos^-$ only detects 14 of them. Specifically, compared with $Chronos^-$, Chronos always achieves more delay block coverage on all four target distributed systems. In total, Chronos covers 1402 more delay blocks, achieving an improvement of 18.10% delay block coverage. Thus, we can conclude that the transient delay helps achieve better performance on both delay block coverage and bug detection. It significantly improves the testing performance, which adequately answers **RQ3**.

6.4. Accuracy of Chronos

To evaluate the **false positives** introduced by transient delays, we first collected all timeout bugs reported by Chronos in the testing phase. Then we reproduce these bugs by replacing the transient delays with actual delays in Chronos. If the bug is reproduced, then it is regarded as the true positive. Otherwise, it is a false positive.

In the testing phases, Chronos reported 28 bugs in total. After the bug reproduction phases, 27 bugs (96.4% of all) are confirmed as true positives. There is only one false positive reported by Chronos. The reason is that in most cases, the timeout handling code l_2 is independent of the code l_1 in actual execution time t, so the transient delay is equivalent to the actual delay and no false positives will be introduced, as we discussed in section 4.4.

However, in some specific cases, the code may have some assumptions about the actual elapsed time. Figure 14 shows the false positive introduced by transient delay. The node sets the timeout value to 10 seconds (line 7) and sends an asynchronous request. If a timeout happens, it releases the resources (line 6). This code has an assumption that the initialize process in line 9 should be completed within 10 seconds. Therefore, the pointer ‘cli’ will be initialized first and this nullptr will not be triggered when the system is running normally. However, if we use the transient delay,

the code in line 13 will be executed immediately before finishing the ‘initialize_resource()’, causing the node to crash. Fortunately, such false positive case is rare and can be easily eliminated by replacing transient delays with actual delays in the reproduction phase.

```
1  bool Node::node_setup(...) {
2     asio::io_context io_context;
3     io_context.set_timeout(seconds(10));
4     io_context.async_wait([&](error_code& err) {
5        if (err == operation_timeout) {
6           release_resource();}});
7     asio::async_request(endpoints,io_context);
8     // assumption: it should be finished in 10s
9     initialize_resource(); //init cli here
10    ...
11 }
12 void Node::release_resource(){
13    this->cli->cancel(); //nullptr
14 }
```

Figure 14. The false positive introduced by transient delay. It can be easily filtered through the reproduction process with actual delays.

False negatives analysis. Library injection in Chronos has an assumption that most code in DSUTs will not directly manipulate IO, but rather through existing common runtime libraries. However, if some codes in DSUTs directly call IO-related system calls with the timeout mechanisms, then Chronos will miss them. Thankfully, such scenarios are rare in real-world distributed system implementation. To evaluate the false negatives introduced by library injection, we manually searched the code sites that directly call system calls with timeout settings in these four distributed systems. In C/C++ language, there are two ways to directly call system calls, using APIs in ‘sys/syscall.h’ or the asm inline assembly. In Java language, they need to use the APIs in the JNA (Java Native Access) library. In Go language, they need to use APIs in the package ‘syscall’. Results show that there are no such code sites. Hence, the false negatives of Chronos are negligible. The accuracy of Chronos is satisfying, which adequately answers **RQ4**.

7. Discussion

More bug types support. Currently, Chronos has supported timeout bug detection by checking whether the function or service of DSUT is available or not. Chronos has already been adapted to four widely used distributed systems and has found 27 previously unknown bugs. However, there are still some other types of bugs, e.g., fail-slow [63] bugs, hidden in the distributed systems.

Fail-slow is a fault where a hardware or software component can still function (does not fail-stop) but in much lower performance than expected. IASO [64] and PERSEUS [65] detect degraded performance by calculating the response ratio and throughput of write as the evaluation metrics. One key insight of Chronos is that it generates much more abnormal delays than those that naturally occur during regular use. This approach is orthogonal to the above works. After executing injected delays for a period of time, Chronos closes all the delays and monitors whether the performance of the system recovers or slows down. The delayed environment provides a unique testing environment that makes it easier to expose fail-slow bugs.

However, different from the timeout bug (server crash or hang) which has a precise and deterministic definition, the fail-slow bug is difficult to define accurately. ”Slow” is a qualitative rather than a quantitative concept, which makes it challenging to establish a precise definition of what constitutes slow behavior. As a result, fail-slow bug testing often produces many false positives. Both IASO and PERSEUS have faced this problem. To address this issue, a reliable and precise oracle for fail-slow bugs needs to be explored in future research.

More Runtime Context. At present, Chronos utilizes the deep-priority guided algorithm to dynamically select delay sequences based on the execution depth of each delay block. By prioritizing the delay sequences that can explore new delay blocks in deep paths, Chronos covers more delay blocks and detects 17 more timeout bugs compared to other methods. While the current deep-priority strategy is powerful, more fine-grained runtime contexts of DSUTs can also be collected as better guidance for the testing process.

For example, the timestamp of each triggered delay can be analyzed in the runtime. And the execution time between each delay block can be calculated. A natural assumption is that the longer time it takes to reach the delay block, the deeper the delay block is. How to effectively model both the execution time and depth of the delay block is worth exploring in the future.

8. Related Work

Fault injection technology: Software fault injection (SFI) [66] is a well-established and commonly utilized technique to identify and mitigate potential system failures by intentionally injecting faults or errors. It holds strong relevance to our work, as Chronos utilizes it for injecting delays. There are two main types of software fault injection in system testing [67]. (1) The first is run-time fault injection, such as chaos engineering induces faults into a running software system at the OS level. Developers in Netflix have developed Chaos Monkey [13] and Simian Army [14] to imitate the complex environment in a cloud system. ChaosBlade [15] from Alibaba supports various experiment-based fault injections such as network fluctuations, disk occupation, and CPU scheduling for testing distributed systems. Jepsen [68] performs fault injection on unmodified distributed data management systems by plenty of manually written test cases. ChaosMachine [69] leverages fault injection to make a live analysis of JVM’s exception-handling functions. However, such injections are coarse-grained and ignore runtime context. (2) The second type is compile-time fault injection [70], [71] that injects some predefined errors into the source code of the software system at specific locations and tests whether it can handle them reasonably during the execution. FlipIt [72], as an example, proposed a fault injector on top of LLVM to enumerate errors in the compile time and activate them

in the runtime. PairCheck [73] designs the fault injection framework to simulate the occasional errors and trigger the error handling codes. Phoenix [74] detects resilience issues in blockchains by injecting context-sensitive chaos. Frameworks like LFI [75], PreFail [76], etc., enable developers to write their own fault injection strategies. However, traditional fault injection tools either inject faults into the environment or directly into the source code. In contrast, Chronos injects delays into the runtime libraries, bringing both precision and generality.

Implementation-Level Model Checking: Distributed system model checkers in implementation-level [7], [8] also simulate failures and enumerate the orders of non-deterministic events to detect bugs, sharing a similarity with our method. For example, CMC [9] is the first model checker that checks C and C++ implementations directly, eliminating the need for a separate abstract description of the system behavior. MODIST [6] systematically simulates a variety of network conditions and failures and repeatedly infers all possible actions of target systems. However, it is difficult for them to automatically identify all timeout mechanisms due to their various implementations in DSUTs. Besides, they suffer from the state space explosion problem due to the huge exploring space in real-world distributed systems. Different from them, Chronos employs deep-priority guided fuzzing to effectively explore the delay combinations and detect timeout bugs.

Mutation-based fuzzing is an effective method to explore testing space and detect bugs in various applications [77], [78], [79]. It is also related to our work since the deep-priority guided fuzzing algorithm is a mutation-based technology. Many fuzzing tools are proposed to test the system faults handling [80], [81]. For example, FIFUZZ [18] mutates combinations of errors based on the code coverage and detects bugs in the error handling. CrashFuzz [25] mutates the crash/reboot sequences by coverage-guided fuzzing and detects recover bugs in cloud systems. However, they do not focus on timeout mechanisms and code coverage-guided methods are inefficient for exploring deep delay blocks.

Main Difference: Different from the above work, Chronos is a general test framework for injecting deep-priority transient delays and detecting timeout bugs in distributed systems. Chronos first proposes the general runtime delayed libraries. Any systems running with them are automatically triggering fine-grained delays. To explore more deep paths, Chronos harnesses the deep-priority delay selector which dynamically prioritizes the delay sequence that can explore delay blocks in deep paths. To accelerate the testing process and mitigate the overhead of actual delays, we propose transient delays that quickly trigger the timeout mechanisms in DSUTs. Furthermore, due to the scalability of the testing framework, Chronos can be quickly adapted to other distributed systems.

9. Conclusion

In this paper, we propose Chronos, an automatic testing framework for detecting timeout bugs in distributed systems based on deep-priority transient delays. Chronos first designs and implements the general delayed libraries that are dynamically linked to the DSUTs during runtime. Then Chronos employs the deep-priority guided fuzzing to explore timeout bugs in deep paths and the transient delays to accelerate the test process. We implement and evaluate Chronos on four widely used distributed systems: ZooKeeper, MySQL-Cluster, HDFS, and Go-Ethereum. The results show that Chronos covers 26.40%, 21.69%, and 15.14% more delay blocks on average compared with other state-of-the-art approaches. Chronos successfully detected 27 previously unknown timeout bugs. Our future work will consider enhancing Chronos with more delay strategies and support more bug types.

10. ACKNOWLEDGEMENTS

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002).

References

[1] G. Le Lann, "Distributed systems-towards a formal approach." in *IFIP congress*, vol. 7, 1977, pp. 155–160.

[2] R. C. Nunes and I. Jansch-Pôrto, "Modeling communication delays in distributed systems using time series," in *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.* IEEE, 2002, pp. 268–273.

[3] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7, no. 10.1145. Portland, OR, 2000, pp. 343 477–343 502.

[4] apache ZooKeeper, "Zookeeper connect timeout," https://www.theverge.com/2020/12/10/22168439/facebook-down-outage-instagram-messenger-whatsapp, 2023, accessed on March 31, 2023.

[5] ZOOKEEPER-3189, "Zookeeper connect timeout," https://issues.apache.org/jira/browse/ZOOKEEPER-3189, 2023, accessed on March 31, 2023.

[6] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "Modist: Transparent model checking of unmodified distributed systems," in *NSDI'09*, 2009, pp. 213–228.

[7] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "{SAMC}:{Semantic-Aware} model checking for fast discovery of deep bugs in cloud systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 399–414.

[8] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa *et al.*, "Flymc: Highly scalable testing of complex interleavings in distributed systems," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[9] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill, "Cmc: A pragmatic approach to model checking real code," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 75–88, 2002.

[10] H. Ziade, R. A. Ayoubi, R. Velazco *et al.*, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[11] M. Eslami, B. Ghavami, M. Raji, and A. Mahani, "A survey on fault injection methods of digital integrated circuits," *Integration*, vol. 71, pp. 154–163, 2020.

[12] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[13] Netflix, "Chaos monkey," https://netflix.github.io/chaosmonkey/, 2022, accessed at December 29, 2022.

[14] A. Netflix, "Simian army," https://github.com/Netflix/SimianArmy, 2022, accessed at December 29, 2022.

[15] chaosblade io, "Chaos blade," https://netflix.github.io/chaosmonkey/, 2022, accessed at December 29, 2022.

[16] chaos mesh org, "Chaos mesh a powerful chaos engineering platform for kubernetes," https://chaos-mesh.org/, 2022, accessed at December 29, 2022.

[17] J.-J. Bai, Y.-P. Wang, H.-Q. Liu, and S.-M. Hu, "Mining and checking paired functions in device drivers using characteristic fault injection," *Information and Software Technology*, vol. 73, pp. 122–133, 2016.

[18] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *the 29th USENIX Security Symposium (Security'20)*, 2020.

[19] R. Banabic and G. Candea, "Fast black-box testing of system recovery code," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 281–294.

[20] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2012.

[21] K. Cong, L. Lei, Z. Yang, and F. Xie, "Automatic fault injection for driver robustness testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 361–372.

[22] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "{FATE} and {DESTINI}: A framework for cloud recovery testing," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[23] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian, "Parametric trojans for fault-injection attacks on cryptographic hardware," in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2014, pp. 18–28.

[24] M. S. R. M. T. Millstein and M. Musuvathi, "Can you fool me? towards automatically checking protocol gullibility."

[25] Y. Gao, W. Dou, D. Wang, W. Feng, J. Wei, H. Zhong, and T. Huang, "Coverage guided fault injection for cloud systems," in *Proceedings of IEEE/ACM SIGSOFT International Conference on Software Engineering (ICSE)*, 2023.

[26] F. Junqueira and B. Reed, *ZooKeeper: distributed process coordination*. " O'Reilly Media, Inc.", 2013.

[27] A. Davies and H. Fisk, *MySQL clustering*. Sams Publishing, 2006.

[28] D. Borthakur *et al.*, "Hdfs architecture guide," *Hadoop apache project*, vol. 53, no. 1-13, p. 2, 2008.

[29] go ethereum, "go-ethereum. official go implementation of the ethereum protocol," https://geth.ethereum.org/, 2022, accessed at December 6, 2022.

[30] V. K. Garg, *Elements of distributed computing*. John Wiley & Sons, 2002.

[31] csis.pace.edu, "communication," http://csis.pace.edu/∼marchese/CS865/Lectures/Chap4/Chapter4.htm, 2023, accessed on March 31, 2023.

[32] ida.liu.se, "Communication in distributed systems," https://www.ida.liu.se/∼TDDD25/lectures/lect3.pdf, 2023, accessed on March 31, 2023.

[33] webstor.srmist.edu.in, "Communication in distributed system," https://www.ida.liu.se/∼TDDD25/lectures/lect3.pdf, 2023, accessed on March 31, 2023.

[34] inf.fu berlin.de, "Indirect communication," http://www.inf.fu-berlin.de/inst/ag-se/teaching/V-NETZPR-2015/07_Indirect_Communication%20-%20I.pdf, 2023, accessed on March 31, 2023.

[35] M. Hedayati and S. Gravani, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019.

[36] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "Pku pitfalls: Attacks on pku-based memory isolation systems," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 1409–1426.

[37] B. H. Tay and A. L. Ananda, "A survey of remote procedure calls," *ACM SIGOPS Operating Systems Review*, vol. 24, no. 3, pp. 68–79, 1990.

[38] A. Venkataraman and K. K. Jagadeesha, "Evaluation of inter-process communication mechanisms," *Architecture*, vol. 86, p. 64, 2015.

[39] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM computing surveys (CSUR)*, vol. 36, no. 4, pp. 335–371, 2004.

[40] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. pearson education, 2005.

[41] J. D. Herbsleb and A. Mockus, "An empirical study of speed and communication in globally distributed software development," *IEEE Transactions on software engineering*, vol. 29, no. 6, pp. 481–494, 2003.

[42] N. M. Josuttis, *SOA in practice: the art of distributed system design*. " O'Reilly Media, Inc.", 2007.

[43] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[44] ludun, "Datastreamer should reset thread interrupted state in createblockoutputstream," https://issues.apache.org/jira/browse/HDFS-15379?page=com.atlassian.jira.plugin.system.issuetabpanels%3Acomment-tabpanel&focusedCommentId=17125997#comment-17125997, 2023.

[45] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007.

[46] A. MySQL, "Mysql," 2001.

[47] Ethereum, "Welcome to ethereum," https://ethereum.org/en/, 2022, accessed at December 23, 2022.

[48] L. WIKI, "netem," https://wiki.linuxfoundation.org/networking/netem, 2023.

[49] S. Koranne and S. Koranne, "Standard libraries," *Handbook of Open Source Tools*, pp. 105–111, 2011.

[50] S. Koranne, "Boost c++ libraries," *Handbook of open source tools*, pp. 127–143, 2011.

[51] american fuzzy lop, "American fuzzy lop user guide," https://afl-1.readthedocs.io/en/latest/user_guide.htmlL, 2023.

[52] G. Cloud, "Setting request timeout (services)," https://cloud.google.com/run/docs/configuring/request-timeout, 2023, accessed at March 30, 2023.

[53] J. Y. Monteith, J. D. McGregor, and J. E. Ingram, "Hadoop and its evolving ecosystem," in *5th International Workshop on Software Ecosystems (IWSECO 2013)*, vol. 50. Citeseer, 2013, p. 74.

[54] A. Lentz and M. AB, "An introduction to mysql cluster architecture and use," *MySQL AB, November*, 2006.

[55] CoinMarketCap, "Coinmarketcap," https://coinmarketcap.com, 2023, accessed on March 31, 2023.

[56] Intel, "Hibench suite," https://github.com/Intel-bigdata/HiBench, 2023, accessed at March 30, 2023.

[57] M. Rigger and Z. Su, "Testing database engines via pivoted query synthesis." in *OSDI*, vol. 20, 2020, pp. 667–682.

[58] Ethereum, https://github.com/drandreaskrueger/chainhammer, 2023.

[59] vdcresearch, "Languaged used in distributed system," https://www.vdcresearch.com/results.html?addsearch=languaged+used+in+distribuetd+systems, 2023, accessed on March 31, 2023.

[60] dzone.com, "Top language for distributed system," https://dzone.com/articles/top-languages-for-distributed-systems, 2023, accessed on March 31, 2023.

[61] jaxenter.com, "Best languages in distributed system," https://jaxenter.com/programming-languages-best-distributed-systems-169281.html, 2023, accessed on March 31, 2023.

[62] I. Docker, "Docker," *lınea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020.

[63] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey *et al.*, "Fail-slow at scale: Evidence of hardware performance faults in large production systems," *ACM Transactions on Storage (TOS)*, vol. 14, no. 3, pp. 1–26, 2018.

[64] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi, "Iaso: A fail-slow detection and mitigation framework for distributed storage services." in *USENIX Annual Technical Conference*, 2019, pp. 47–62.

[65] R. Lu, E. Xu, Y. Zhang, F. Zhu, Z. Zhu, M. Wang, Z. Zhu, G. Xue, J. Shu, M. Li *et al.*, "Perseus: A {Fail-Slow} detection framework for cloud storage systems," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 49–64.

[66] J. M. Voas and G. McGraw, *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., 1997.

[67] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[68] Jepsen, "Distributed systems safety research," http://jepsen.io/, 2023.

[69] L. Zhang, B. Morin, P. Haller, B. Baudry, and M. Monperrus, "A chaos engineering system for live analysis and falsification of exception-handling in the jvm," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2534–2548, 2019.

[70] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1466–1482.

[71] J. Gravellier, J.-M. Dutertre, Y. Teglia, and P. L. Moundi, "Faultline: Software-based fault injection on memory transfers," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2021, pp. 46–55.

[72] J. Calhoun, L. Olson, and M. Snir, "Flipit: An llvm based fault injector for hpc," 08 2014, pp. 547–558.

[73] J.-J. Bai, Y.-P. Wang, H.-Q. Liu, and S.-M. Hu, "Mining and checking paired functions in device drivers using characteristic fault injection," *Information and Software Technology*, vol. 73, pp. 122–133, 2016.

[74] F. Ma, Y. Chen, Y. Zhou, J. Sun, Z. Su, Y. Jiang, J. Sun, and H. Li, "Phoenix: Detect and locate resilience issues in blockchain via context-sensitive chaos," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1182–1196.

[75] P. D. Marinescu and G. Candea, "Efficient testing of recovery code using fault injection," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 4, pp. 1–38, 2011.

[76] P. Joshi, H. S. Gunawi, and K. Sen, "Prefail: A programmable tool for multiple-failure injection," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 171–188.

[77] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "{EnFuzz}: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1967–1983.

[78] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: packet sequence oriented fuzzing for protocol implementations," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4481–4498.

[79] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "Healer: Relation learning guided kernel fuzzing," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 344–358.

[80] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2517–2532.

[81] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, "Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols," in *Proceedings 2023 Network and Distributed System Security Symposium*, 2023.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

The paper introduces Chronos, a framework to find timeout bugs in distributed application. Chronos uses a new concept called deep-priority search to find ideal locations for injecting delays into the system under test. Delays are injected by manipulating the system libraries used by the target applications. To reduce the time overhead in each fuzzing cycle, Chronos sets delays in the target application to zero and executes it with the timing-manipulated application. The authors name this technique transient delays. With Chronos approach, the authors found 27 timeout bugs in real-world applications, all of which were confirmed and fixed by the application developers. The approach is compared to other state-of-the-art techniques, including coverage-guided fault injection, and outperforms all of them significantly on the four chosen target applications.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction

A.3. Reasons for Acceptance

1) The work applies a known technique, fault injection, in a novel way to find vulnerabilities in distributed applications.
2) The work offers a new technique for finding the ideal injection points for revealing bugs.
3) The work introduces the new concept of transient delays.
4) The implemented system found an impressive list of bugs.
5) The work provides tools implementing these techniques.