Understanding, Detecting and Localizing Partial Failures in Large System Software

Chang Lou, Peng Huang, and Scott Smith, *Johns Hopkins University*

https://www.usenix.org/conference/nsdi20/presentation/lou

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20) is sponsored by

Understanding, Detecting and Localizing Partial Failures in Large System Software

Chang Lou
Johns Hopkins University

Peng Huang
Johns Hopkins University

Scott Smith
Johns Hopkins University

Abstract

Partial failures occur frequently in cloud systems and can cause serious damage including inconsistency and data loss. Unfortunately, these failures are not well understood. Nor can they be effectively detected. In this paper, we first study 100 real-world partial failures from five mature systems to understand their characteristics. We find that these failures are caused by a variety of defects that require the unique conditions of the production environment to be triggered. Manually writing effective detectors to systematically detect such failures is both time-consuming and error-prone. We thus propose OmegaGen, a static analysis tool that automatically generates customized watchdogs for a given program by using a novel program reduction technique. We have successfully applied OmegaGen to six large distributed systems. In evaluating 22 real-world partial failure cases in these systems, the generated watchdogs can detect 20 cases with a median detection time of 4.2 seconds, and pinpoint the failure scope for 18 cases. The generated watchdogs also expose an unknown, confirmed partial failure bug in the latest version of ZooKeeper.

1 Introduction

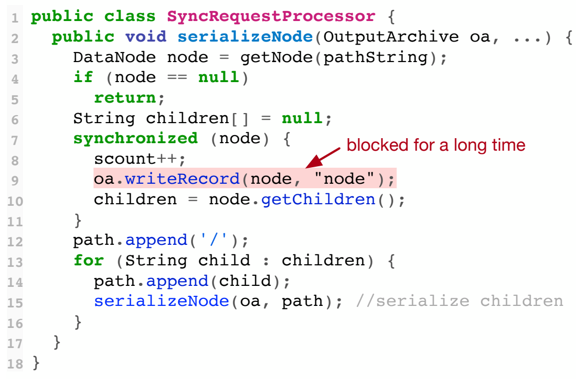
It is elusive to build large software that never fails. Designers of robust systems therefore must devise runtime mechanisms that proactively check whether a program is still functioning properly, and react if not. Many of these mechanisms are built with a simple assumption that when a program fails, it fails completely via crash, abort, or network disconnection.

This assumption, however, does not reflect the complex failure semantics exhibited in modern cloud infrastructure. A typical cloud software program consists of tens of modules, hundreds of dynamic threads, and tens of thousands of functions for handling different requests, running various background tasks, applying layers of optimizations, *etc.* Not surprisingly, such a program in practice can experience *partial failures*, where some, but not all, of its functionalities are broken. For example, for a data node process in a modern distributed file system, a partial failure could occur when a rebalancer thread within this process can no longer distribute unbalanced blocks to other remote data node processes, even though this process is still alive. Or, a block receiver daemon in this data node process silently exits, so the blocks are no longer persisted to disk. These partial failures are *not* a latent problem that operators can ignore; they can cause serious damage including inconsistency, "zombie" behavior and data loss. Indeed, partial failures are behind many catastrophic real-world outages [1, 17, 39, 51, 52, 55, 66, 85, 86]. For example, Microsoft Office 365 mail service suffered an 8-hour outage because an anti-virus engine module of the mail server was stuck in identifying some suspicious message [39].

When a partial failure occurs, it often takes a long time to detect the incident. In contrast, a process suffering a total failure can be quickly identified, restarted or repaired by existing mechanisms, thus limiting the failure impact. Worse still, partial failures cause mysterious symptoms that are incredibly difficult to debug [78], e.g., `create()` requests time out but `write()` requests still work. In a production ZooKeeper outage due to the leader failing partially [86], even after an alert was triggered, the leader logs contained few clues about what went wrong. It took the developer significant time to localize the fault within the problematic leader process (Figure 1). Before pinpointing the failure, a simple restart of the leader process was fruitless (the symptom quickly re-appeared).

Both practitioners and the research community have called attention to this gap. For example, the Cassandra developers adopted the more advanced accrual failure detector [73], but still conclude that its current design *"has very little ability to effectively do something non-trivial to deal with partial failures"* [13]. Prabhakaran *et al.* analyze partial failure specific to disks [88]. Huang *et al.* discuss the gray failure [76] challenge in cloud infrastructure. The overall characteristics of software partial failures, however, are not well understood.

In this paper, we first seek to answer the question, *how do partial failures manifest in modern systems?* To shed some light on this, we conducted a study (Section 2) of 100 real-world partial failure cases from five large-scale, open-source systems. We find that nearly half (48%) of the studied failures



```
1  public class SyncRequestProcessor {
2    public void serializeNode(OutputArchive oa, ...) {
3      DataNode node = getNode(pathString);
4      if (node == null)
5        return;
6      String children[] = null;
7      synchronized (node) {
8        scount++;
9        oa.writeRecord(node, "node");
10       children = node.getChildren();
11     }
12     path.append('/');
13     for (String child : children) {
14       path.append(child);
15       serializeNode(oa, path); //serialize children
16     }
17   }
18 }
```

Figure 1: A production ZooKeeper outage due to partial failure [86].

cause certain software-specific functionality to be stuck. In addition, the majority (71%) of the studied failures are triggered by unique conditions in a production environment, *e.g.*, bad input, scheduling, resource contention, flaky disks, or a faulty remote process. Because these failures impact internal features such as compaction and persistence, they can be unobservable to external detectors or probes.

How to systematically detect and localize partial failures at runtime? Practitioners currently rely on running *ad-hoc* health checks (*e.g.*, send an HTTP request every few seconds and check its response status [3, 42]). But such health checks are too shallow to expose a wide class of failures. The state-of-the-art research work in this area is Panorama [75], which converts various requestors of a target process into observers to report gray failures of this process. This approach is limited by what requestors can observe externally. Also, these observers cannot localize a detected failure within the faulty process.

We propose a novel approach to construct effective partial failure detectors through *program reduction*. Given a program P, our basic idea is to derive from P a reduced but representative version W as a detector module and periodically test W in production to expose various potential failures in P. We call W an *intrinsic watchdog*. This approach offers two main benefits. First, as the watchdog is derived from and “imitates” the main program, it can more accurately reflect the main program’s status compared to the existing stateless heartbeats, shallow health checks or external observers. Second, reduction makes the watchdog succinct and helps localize faults.

Manually applying the reduction approach on large software is both time-consuming and error-prone for developers. To ease this burden, we design a tool, *OmegaGen*, that statically analyzes the source code of a given program and generates customized intrinsic watchdogs for the target program.

Our insight for realizing program reduction in OmegaGen is that W’s goal is *solely* to detect and localize runtime errors; therefore, it does not need to recreate the full details of P’s business logic. For example, if P invokes `write()` in a tight loop, for checking purposes, a W with one `write()` may be sufficient to expose a fault. In addition, while it is tempting to check all kinds of faults, given the limited resources, W should focus on checking faults manifestable only in a production environment. Logical errors that deterministically lead to wrong results (*e.g.*, incorrect sorting) should be the focus of offline unit testing. Take Figure 1 as an example. In checking the `SyncRequestProcessor`, W need not check most of the instructions in function `serializeNode`, *e.g.*, lines 3–6 and 8. While there might be a slim chance these instructions would also fail in production, repeatedly checking them would yield diminishing returns for the limited resource budget.

Accurately distinguishing logically-deterministic faults and production-dependent faults in general is difficult. OmegaGen uses heuristics to analyze how “vulnerable” an instruction is based on whether the instruction performs some I/O, resource allocation, async wait, *etc.* So since line 9 of Figure 1 performs a write, it would be assessed as vulnerable and tested in W. It is unrealistic to expect W to always include the failure root cause instruction. Fortunately, a ballpark assessment often suffices. For instance, even if we only assess that the entire `serializeNode` function or its caller is vulnerable, and periodically test it in W, W can still detect this partial failure.

Once the vulnerable instructions are selected, OmegaGen will encapsulate them into checkers. OmegaGen’s second contribution is providing several strong isolation mechanisms so the watchdog checkers do not interfere with the main program. For memory isolation, OmegaGen identifies the *context* for a checker and generates context managers with hooks in the main program which replicates contexts before using them in checkers. OmegaGen removes side-effects from I/O operations through redirection and designs an idempotent wrapper mechanism to safely test non-idempotent operations.

We have applied OmegaGen to six large (28K to 728K SLOC) systems. OmegaGen automatically generates tens to hundreds of watchdog checkers for these systems. To evaluate the effectiveness of the generated watchdogs, we reproduced 22 **real-world** partial failures. Our watchdogs can detect 20 cases with a median detection time of 4.2 seconds and localize the failure scope for 18 cases. In comparison, the best manually written baseline detector can only detect 11 cases and localize 8 cases. Through testing, our watchdogs exposed a new, confirmed partial failure bug in the latest ZooKeeper.

2 Understanding Partial Failures

Partial failures are a well known problem. Gupta and Shute report that partial failures occur much more commonly than total failures in the Google Ads infrastructure [70]. Researchers studied partial disk faults [88] and slow hardware faults [68]. But how software fails partially is not well understood. In this Section, we study real-world partial failures to gain insight into this problem and to guide our solution design.

Scope We focus on partial failure at the process granularity. This process could be standalone or one component in a large service (*e.g.*, a datanode in a storage service). Our studied partial failure is with respect to a process deviating from the functionalities it is supposed to provide *per se*, *e.g.*, store and

Software	Lang.	Cases	Ver.s (Range)	Date Range
ZooKeeper	Java	20	17 (3.2.1–3.5.3)	12/01/2009–08/28/2018
Cassandra	Java	20	19 (0.7.4–3.0.13)	04/22/2011–08/31/2017
HDFS	Java	20	14 (0.20.1–3.1.0)	10/29/2009–08/06/2018
Apache	C	20	16 (2.0.40–2.4.29)	08/02/2002–03/20/2018
Mesos	C++	20	11 (0.11.0–1.7.0)	04/08/2013–12/28/2018

Table 1: Studied software systems, the partial failure cases, and the unique versions, version and date ranges these cases cover.

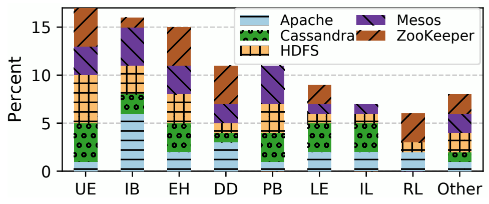


Figure 2: Root cause distribution. *UE*: uncaught error; *IB*: indefinite blocking; *EH*: buggy error handling; *DD*: deadlock; *PB*: performance bug; *LE*: logic error; *IL*: infinite loop; *RL*: resource leak.

balance data blocks, whether it is a service component or a standalone server. We note that users may define a partial failure at the service granularity (*e.g.*, Google drive becomes read-only), the underlying root cause of which could be either some component crashing or failing partially.

Methodology We study five large, widely-used software systems (Table 1). They provide different services and are written in different languages. To collect the study cases, we first crawl all bug tickets tagged with critical priorities in the official bug trackers. We then filter tickets from testing and randomly sample the remaining failures tickets. To minimize bias in the types of partial failures we study, we exhaustively examining each sampled case and manually determine whether it is a complete failure (*e.g.*, crash), and discard if so. In total, we collected 100 failure cases (20 cases for each system).

2.1 Findings

Finding 1: *In all the five systems, partial failures appear throughout release history (Table 1). 54%*[1] *of them occur in the most recent three years' software releases.*

Such a trend occurs in part because as software evolves, new features and performance optimizations are added, which complicates the failure semantics. For example, HDFS introduced a short-circuit local reads feature [30] in version 0.23. To implement this feature, a `DomainSocketWatcher` was added that watches a set of Unix domain sockets and invokes a callback when they become readable. But this new module can accidentally exit in production and cause applications performing short-circuit reads to hang [29].

Finding 2: *The root causes of studied failures are diverse. The top three (total 48%) root cause types are uncaught errors, indefinite blocking, and buggy error handling (Figure 2).*

Uncaught error means certain operation triggers some error condition that is not expected by the software. As an example, the streaming session in Cassandra could hang when the stream reader encounters errors other than `IOException` like `RuntimeException` [6]. Indefinite blocking occurs when some function call is blocked forever. In one case [27], the `EditLogTailer` in a standby HDFS namenode made an RPC `rollEdits()` to the active namenode; but this call was blocked when the active namenode was frozen but not crashed, which prevented the standby from becoming active. Buggy error handling includes silently swallowing errors, empty handlers [93], premature continuing, *etc.* Other common root causes include deadlock, performance bugs, infinite loop and logic errors.

[1]With sample size 100, the percents also represent the absolute numbers.

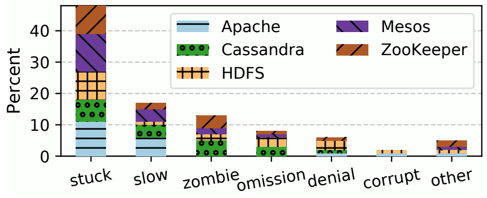


Figure 3: Consequence of studied failures.

Finding 3: *Nearly half (48%) of the partial failures cause some functionality to be stuck.*

Figure 3 shows the consequences of the studied failures. Note that these failures are all partial. For the *"stuck"* failures, some software module like the socket watcher was not making any progress; but the process was not completely unresponsive, *i.e.*, its heartbeat module can still respond *in time*. It may also handle other requests like non-local reads.

Besides "stuck" cases, 17% of the partial failures causes certain operation to take a long time to complete (the *"slow"* category in Figure 3). These slow failures are not just inefficiencies for optional optimization. Rather, they are severe performance bugs that cause the affected feature to be barely usable. In one case [5], after upgrading Cassandra 2.0.15 to 2.1.9, users found the read latency of the production cluster increased from 6 ms/op to more than 100 ms/op.

Finding 4: *In 13% of the studied cases, a module became a "zombie" with undefined failure semantics.*

This typically happens when the faulty module accidentally exits its normal control loop or it continues to execute even when it encounters some severe error that it cannot tolerate. For example, an unexpected exception caused the ZooKeeper listener module to accidentally exit its `while` loop so new nodes could no longer join the cluster [46]. In another case, the HDFS datanode continued even if the block pool failed to initialize [26], which would trigger a `NullPointerException` whenever it tried to do block reports.

Finding 5: *15% of the partial failures are silent (including data loss, corruption, inconsistency, and wrong results).*

They are usually hard to detect without detailed correctness specifications. For example, when the Mesos agent garbage collects old slave sandboxes, it could incorrectly wipe out the persistent volume data [37]. In another case [38], the Apache web server would “go haywire”, *e.g.*, a request for a .js file would receive a response of image/png, because the backend connections are not properly closed in case of errors.

Finding 6: *71% of the failures are triggered by some specific environment condition, input, or faults in other processes.*

For example, a partial failure in ZooKeeper can only be triggered when some corrupt message occurs in the `length` field of a record [66]. Another partial failure in the ZooKeeper leader would only occur when a connecting follower hangs [50], which prevents other followers from joining the cluster. These partial failures are hard to be exposed by pre-production testing and require mechanisms to detect at runtime. Moreover, if a runtime detector uses a different setup or checking input, it may not detect such failures.

Finding 7: *The majority (68%) of the failures are “sticky”.*

Sticky means the process will not recover from the faults by itself. The faulty process needs to be restarted or repaired to function again. In one case, a race condition caused an unexpected `RejectedExecutionException`, which caused the RPC server thread to silently exit its loop and stop listening for connections [9]. This thread must be restarted to fix the issue. For certain failures, some extra repair actions such as fixing a file system inconsistency [25] are needed.

The remaining (32%) failures are “transient”, *i.e.*, the faulty modules could possibly recover after certain condition changes, *e.g.*, when the frozen namenode becomes responsive [27]. However, these non-sticky failures already incurred damage for a long time by then (15 minutes in one case [45]).

Finding 8: *The median diagnosis time is 6 days and 5 hours.*

For example, diagnosing a Cassandra failure [10] took the developers almost two days. The root cause turned out to be relatively simple: the `MeteredFlusher` module was blocked for several minutes and affected other tasks. One common reason for the long diagnosis time despite simple root causes is that the confusing symptoms of the failures mislead the diagnosis direction. Another common reason is the insufficient exposure of runtime information in the faulty process. Users have to enable debug logs, analyze heap, and/or instrument the code, to identify what was happening during the production failure.

2.2 Implications

Overall, our study reveals that partial failure is a common and severe problem in large software systems. Most of the studied failures are production-dependent (finding 6), which require runtime mechanisms to detect. Moreover, if a runtime detector can localize a failure besides mere detection, it will reduce the difficulty of offline diagnosis (finding 8). Existing detectors such as heartbeats, probes [69], or observers [75] are ineffective because they have little exposure to the affected functionalities internal in a process (e.g., compaction).

One might conclude that the onus is on the developers to add effective runtime checks in their code, such as a timer check for the `rollEdits()` operation in the aforementioned HDFS failure [27]. However, simply relying on developers to anticipate and add defensive checks for every operation is unrealistic. We need a *systematic* approach to help developers construct software-specific runtime checkers.

It would be desirable to completely automate the construction of customized runtime checkers, but this is extremely difficult in the general case given the diversity (finding 2) of partial failures. Indeed, 15% of the studied failures are silent, which require detailed correctness specifications to catch. Fortunately, the majority of failures in our study violate liveness (finding 3) or trigger explicit errors at certain program points, which suggests that detectors can be automatically constructed without deep semantic understanding.

3 Catching Partial Failures with Watchdogs

We consider a large server process π composed of many smaller modules, providing a set of functionalities R, *e.g.*, a datanode server with request listener, snapshot manager, cache manager, *etc.* A failure detector is needed to monitor the process for high availability. We target specifically partial failures. We define a partial failure in a process π to be when a fault does not crash π but causes safety or liveness violation or severe slowness for some functionality $R_f \subsetneq R$. Besides detecting a failure, we aim to localize the fault within the process to facilitate subsequent troubleshooting and mitigation.

Guided by our study, we propose an *intersection principle* for designing effective partial failure detectors—construct customized checks that intersect with the execution of a monitored process. The rationale is that partial failures typically involve specific software feature and bad state; to expose such failures, the detector need to exercise specific code regions with carefully-chosen payloads. The checks in existing detectors including heartbeat and HTTP tests are too generic and too *disjoint* with the monitored process’ states and executions.

We advocate an **intrinsic watchdog** design (Figure 4) that follows the above principle. An intrinsic watchdog is a dedicated monitoring extension for a process. This extension regularly executes a set of checkers tailored to different modules. A watchdog driver manages the checker scheduling and execution, and optionally applies a recovery action. The key objective for detection is to let the watchdog experience similar faults as the main program. This is achieved through (a) executing *mimic-style* checkers (b) using *stateful* payloads (c) sharing execution environment of the monitored process.

Mimic Checkers. Current detectors use two types of checkers: *probe* checkers, which periodically invoke some APIs; *signal* checkers, which monitor some health indicator. Both are lightweight. But a probe checker can miss many failures because a large program has numerous APIs and partial failures may be unobservable at the API level. A signal checker is susceptible to environment noises and usually has poor accuracy. Neither can localize a detected failure.

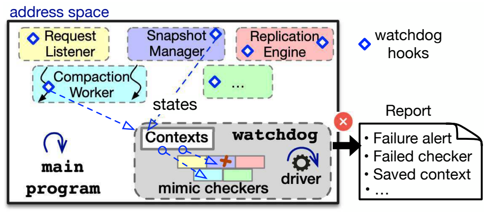


Figure 4: An intrinsic watchdog example.

We propose a more powerful *mimic-style* checker. Such checker selects some representative operations from each module of the main program, imitates them, and detects errors. This approach increases coverage of checking targets. And because the checker exercises code logic similar to the main program in production environment, it can accurately reflect the monitored process' status. In addition, a mimic checker can pinpoint the faulty module and failing instruction.

Synchronized States. Exercising checkers requires payloads. Existing detectors use *synthetic* input (e.g., fixed URLs [3]) or a tiny portion of the program state (e.g., heartbeat variables) as the payload. But triggering partial failures usually entails specific input and program state (§2). The watchdog should exercise its checkers with non-trivial state from the main program for higher chance of exposing partial failures.

We introduce *contexts* in watchdogs. A context is bound to each checker and holds *all* the arguments needed for the checker execution. Contexts are synchronized with the program state through hooks in the main program. When the main program execution reaches a hook point, the hook uses the current program state to update its context. The watchdog driver will not execute a checker unless its context is ready.

Concurrent Execution. It is natural to insert checkers directly in the main program. However, in-place checking poses an inherent tension—on the one hand, catching partial failures requires adding comprehensive checkers; on the other hand, partial failures only occur rarely, but more checkers would slow down the main program in normal scenarios. In-place checkers could also easily interfere with the main program through modifying the program states or execution flow.

We advocate watchdog to run *concurrently* with the main program. Concurrent execution allows checking to be decoupled so a watchdog can execute comprehensive checkers without delaying the main program during normal executions. Indeed, embedded systems domain has explored using concurrent watchdog co-processor for efficient error detection [84]. When a checker triggers some error, the watchdog also will not unexpectedly alter the main program execution. The concurrent watchdog should still live in the same address space to maximize mimic execution and expose similar issues, *e.g.*, all checkers timed out when the process hits long GC pause.

4 Generating Watchdogs with OmegaGen

It is tedious to manually write effective watchdogs for large programs, and it is challenging to get it right. Incautiously written watchdogs can miss checking important functions, alter the main execution, invoke dangerous operations, corrupt program states, *etc.* a watchdog must also be updated as the software evolves. To ease developers' burden, we design a tool, OmegaGen, which uses a novel *program reduction* approach to automatically generate watchdogs described in Section 3. The central challenge of OmegaGen is to ensure the generated watchdog accurately reflects the main program status without introducing significant overhead or side effects.

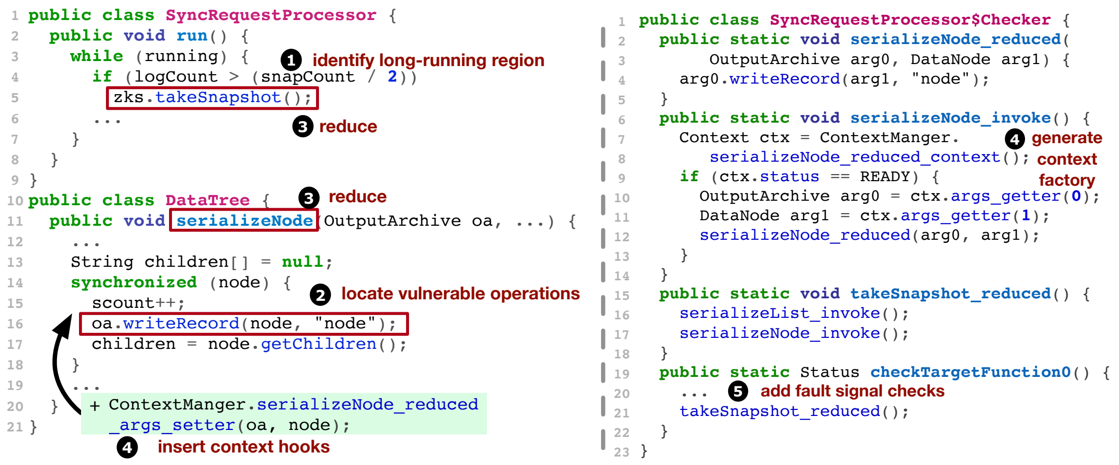
Overview and Target. OmegaGen takes the source code of a program P as an input. It finds the long-running code regions in P and then identifies instructions that may encounter production-dependent issues using heuristics and optional, user-provided annotations. OmegaGen encapsulates the vulnerable instructions into executable checkers and generates watchdog W. It also inserts watchdog hooks in P to update W's contexts and packages a driver to execute W in P. Figure 5 shows an overview example of running OmegaGen.

As discussed in Section 2.2, it is difficult to automatically generate detectors that can catch all types of partial failures. Our approach targets partial failures that surface through explicit errors, blocking or slowness at certain instruction or function in a program. The watchdogs OmegaGen generates are particularly effective in catching partial failures in which some module becomes stuck, very slow or a "zombie" (*e.g.*, the HDFS `DomainSocketWatcher` thread accidentally exiting and affecting short-circuit reads). They are in general ineffective on *silent* correctness errors (*e.g.*, Apache web-server incorrectly re-using stale connections).

4.1 Identify Long-running Methods

OmegaGen starts its static analysis by identifying long-running code regions in a program (step ❶), because watchdogs only target checking code that is continuously executed. Many code regions in a server program are only for one-shot tasks such as database creation, and should be excluded from watchdogs. Some tasks are also either periodically executed such as snapshot or only activated under specific conditions. We need to ensure the activation of generated watchdog is aligned with the life span of its checking target in the main program. Otherwise, it could report wrong detection results.

OmegaGen traverses each node in the program call graph. For each node, it identifies potentially long-running loops in the function body, *e.g.*, `while(true)` or `while(flag)`. Loops with fixed iterations or that iterate over collections will be skipped. OmegaGen then locates all the invocation instructions in the identified loop body. The invocation targets are colored. Any methods invoked by a colored node are also recursively colored. Besides loops, we also support coloring periodic task methods scheduled through common libraries like `ExecutorService` in Java concurrent package. Note that this step may over-extract (*e.g.*, an invocation under a conditional). This is not an issue because the watchdog driver will check context validity at runtime (§4.4).



```java
public class SyncRequestProcessor {
  public void run() {
    while (running) {
      if (logCount > (snapCount / 2))
        zks.takeSnapshot();
      ...
    }
  }
}
public class DataTree {
  public void serializeNode(OutputArchive oa, ...) {
    ...
    String children[] = null;
    synchronized (node) {
      scount++;
      oa.writeRecord(node, "node");
      children = node.getChildren();
    }
    ...
  }  + ContextManger.serializeNode_reduced
       _args_setter(oa, node);
}
```

❶ identify long-running region
❸ reduce
❸ reduce
❷ locate vulnerable operations
❹ insert context hooks

(a) A module in main program

```java
public class SyncRequestProcessor$Checker {
  public static void serializeNode_reduced(
        OutputArchive arg0, DataNode arg1) {
    arg0.writeRecord(arg1, "node");
  }
  public static void serializeNode_invoke() {
    Context ctx = ContextManger.
        serializeNode_reduced_context();
    if (ctx.status == READY) {
      OutputArchive arg0 = ctx.args_getter(0);
      DataNode arg1 = ctx.args_getter(1);
      serializeNode_reduced(arg0, arg1);
    }
  }
  public static void takeSnapshot_reduced() {
    serializeList_invoke();
    serializeNode_invoke();
  }
  public static Status checkTargetFunction0() {
    ...
    takeSnapshot_reduced();
  }
}
```

(b) Generated checker

Figure 5: Example of watchdog checker OmegaGen generated for a module in ZooKeeper.

A complication arises when a method has multiple call-sites, some of which are colored while others are not. Whether this method is long running or not depends on the specific execution. Moreover, an identified long-running loop may turn out to be short-lived in an actual run. To accurately capture the method life span and control the watchdog activation, OmegaGen designs a *predicate*-based algorithm. A *predicate* is a runtime property associated with a method which tracks whether a call site of this method is in fact reached.

For an invocation target inside a potentially long-running loop, a hook is inserted before the loop that sets its predicate and another hook after the loop that unsets its predicate. A callee of a potentially long-running method will have a predicate set to be equal to this caller's predicate. At runtime, the predicates are assigned and evaluated that activates or deactivates the associated watchdog. The predicate instrumentation occurs after OmegaGen finishes the vulnerable operation analysis (§4.2) and program reduction (§4.3).

4.2 Locate Vulnerable Operations

OmegaGen then analyzes the identified long-running methods and further narrows down the checking target candidates (step ❷). This is because even in those limited number of methods, a watchdog cannot afford to check all of their operations. Our study shows that the majority of partial failures are triggered by unique environment conditions or workloads. This implies that operations whose safety or liveness are heavily influenced by its execution environment deserve particular attention. In contrast, operations whose correctness is logically deterministic (*e.g.*, sorting), are better checked through offline testing or in-place assertions. Continuously monitoring such operations inside a watchdog would yield diminishing returns.

OmegaGen uses heuristics to determine for a given operation how *vulnerable* this operation is in its execution environment. Currently, the heuristics consider operations that perform synchronization, resource allocation, event polling, async waiting, invocation with external input argument, file or network I/O as highly vulnerable. OmegaGen identifies most of them through standard library calls. Functions containing complex while loop conditions are considered vulnerable due to potential infinite looping. Simple operations such as arithmetic, assignments, and data structure field accesses are tagged as not vulnerable. In the Figure 5a example, OmegaGen considers the `oa.writeRecord` to be highly vulnerable because its body invokes several write calls. These heuristics are informed by our study but can be customized through a rule table configuration in OmegaGen. For example, we can configure OmegaGen to consider functions with several exception signatures as vulnerable (i.e., potentially improperly handled). We also allow developers to annotate a method with a `@vulnerable` tag in the source code. OmegaGen will locate calls to the annotated method and treat them as vulnerable.

Neither our heuristics nor human judgment can guarantee that the vulnerable operation criteria are always sound and complete. If OmegaGen incorrectly assesses a safe operation as vulnerable, the main consequence is that the watchdog would waste resources monitoring something unnecessarily. Incorrectly assessing a vulnerable operation as risk-free is more concerning. But one nice characteristic of vulnerable operations is that they often propagate [67] – an instruction that blocks indefinitely would also cause its enclosing function to block; and, an instruction that triggers some uncaught error also propagates through the call stack. For example, in a real-world partial failure in ZooKeeper [66], even if OmegaGen misses the exact vulnerable instruction `readString`, a watchdog still has a chance to detect the partial failure if `dserialize` or even `pRequest` is assessed to be vulnerable. On the other hand, if a vulnerable operation is too high-level (*e.g.*, `main` is considered vulnerable), error signals can be swallowed internally and it would also make localizing faults hard.

4.3 Reduce Main Program

With the identified long-running methods and vulnerable operations, OmegaGen performs a top-down program reduction (step ❸) starting from the entry point of long-running methods. For example, in Figure 5a, OmegaGen will try to reduce the `takeSnapshot` function first. When walking the control flow graph of a method to be reduced, if an instruction is tagged as potentially vulnerable, it would be retained in the reduced method. Otherwise, it would be excluded. For a call instruction that is not tagged as vulnerable yet, it would be temporarily retained and OmegaGen will recursively try to reduce the target function. If eventually the body of a reduced method is empty, *i.e.,* no vulnerable operation exists, it will be discarded. Any call instructions that call this discarded method and were temporarily retained are also discarded.

The resulting reduced program not only contains all vulnerable operations reachable from long-running methods but also preserves the original structure, i.e., for a call chain `f` $\hookrightarrow$ `g` $\hookrightarrow$ `h` in the main program, the reduced call chain is `f'` $\hookrightarrow$ `g'` $\hookrightarrow$ `h'` This structure can help localize a reported issue. In addition, when later a watchdog invokes a validator (§4.6), the structure provides information on which validator to invoke.

If a type of vulnerable operation (*e.g.*, the `writeRecord` call in Figure 5a) is included multiple times in the reduced program, it could be redundant in terms of exposing failures. Therefore, OmegaGen will further reduce the vulnerable operations based on whether they have been included already. However, the same type of vulnerable operation may be invoked quite differently in different places, and only a particular invocation would trigger failure. If we are too aggressive in reducing based on occurrences, we may miss the fault-triggering invocation. So, by default OmegaGen only performs intra-procedural occurrence reduction: multiple `writeRecord` calls will not occur within a single reduced method but may occur across different reduced methods.

4.4 Encapsulate Reduced Program

OmegaGen will encapsulate the code snippets retained after step ❸ into watchdogs. But these code snippets may not be directly executable because of missing definitions or payloads. For example, the reduced version of `serializeNode` in Figure 5a contains an operation `oa.writeRecord(node, "node")`. But `oa` and `node` are undefined. OmegaGen analyzes *all* the arguments required for the execution of a reduced method. For each undefined variable, OmegaGen adds a local variable definition at the beginning of the reduced method. It further generates a *context factory* that provides APIs to manage all the arguments for the reduced method (step ❹). Before a variable's first usage in the reduced method, a getter call to the context factory is added to retrieve the latest value at runtime.

To synchronize with the main program, OmegaGen inserts hooks that call setter methods of the same context factory in the (non-reduced) method in the original program at the same point of access. The context hooks are further conditioned on the long-running predicate for this method (§4.1). When the watchdog driver executes a reduced method, it first checks whether the context is ready and skips the execution if the context is not ready. Together, context and predicate control the activation of watchdog checkers—only when the original program reaches the context hooks and the method is truly long-running would the corresponding operation be checked. For example, in the while loop of Figure 5a, if the log count has not reached the snapshot threshold yet, the predicate for `takeSnapshot` is true but the context for the reduced `serializeNode` is not ready so the checking is skipped.

4.5 Add Checks to Catch Faults

After step ❹, the encapsulated reduced methods can be executed in a watchdog. OmegaGen will then add checks for the watchdog driver to catch the failure signals from the execution of vulnerable operations in the reduced methods. OmegaGen targets both liveness and safety violations. Liveness checks are relatively straightforward to add. OmegaGen inserts a timer before running a checker. Setting good timeouts for distributed systems is a well-known hard problem. Prior work [82] argues that replacing end-to-end timeouts with fine-grained timeouts for local operations makes the setting less sensitive. We made similar observations and use a conservative timeout (default 4 seconds). Besides timeouts, the watchdog driver also records the moving average of checker execution latencies to detect potential slow faults.

To detect safety violations, OmegaGen relies on the vulnerable operations to emit explicit error signals (assertions, exceptions, and error codes) and installs handlers to capture them. OmegaGen also captures runtime errors, *e.g.*, null pointer exception, out of memory errors, `IllegalStateException`.

Correctness violations are harder to check automatically without understanding the semantics of the vulnerable operations. Fortunately such silent violations are not very common in our studied cases (§2). Nevertheless, OmegaGen provides a `wd_assert` API for developers to conveniently add semantic checks. When OmegaGen analyzes the program, it will treat `wd_assert` instructions as special vulnerable operations. It performs similar checker encapsulation (§4.4) by analyzing the context needed for such operations and generates checkers containing the `wd_assert` instructions. The original `wd_assert` in the main program will be rewritten as a no-op. In this way, developers can leverage the OmegaGen framework to perform concurrent expensive checks (*e.g.*, if the hashes of new blocks match their checksums) without blocking the main execution.

The watchdog driver records any detected error in a log file. The reported error contains the timestamp, failure type and symptom, failed checker, the corresponding main program location that the failed checker is testing. backtrace, *etc*. The watchdog driver also saves the context used by the failed checker to ease subsequent offline troubleshooting.

4.6 Validate Impact of Caught Faults

An error reported by a watchdog checker could be transient or tolerable. To reduce false alarms, the watchdog runs a validation task after detecting an error. The default validation is to simply re-execute the checker and compare, which is effective for transient errors. Validating tolerable errors requires testing software features. Note that the validator is *not* for handling errors but rather confirming impact. Writing such validation tasks mainly involves invoking some entry functions, *e.g.*, `processRequest(req)`, which is straightforward.

OmegaGen provides skeletons of validation tasks, and currently relies on manual effort to fill out the skeletons. But OmegaGen automates the decision of choosing which validation task to invoke based on which checker failed. Specifically, for a filled validation task T that invokes a function `f` in the main program, OmegaGen searches the generated reduced program structure (§4.3) in topological order and tries to find the first reduced method `m'` that either matches `f` or any method in the `f`'s callgraph. Then OmegaGen generates a hashmap that maps all the checkers that are rooted under `m'` to task T. At runtime, when an error is reported, the watchdog driver checks the map to decide which validator to invoke.

4.7 Prevent Side Effects

Context Replication. To prevent the watchdog checkers from accidentally modifying the main program's states, OmegaGen analyzes *all* the variables (context) referenced in a checker. It generates a replication setter in the checker's context manager, which will replicate the context when invoked. The replication ensures any modifications are contained in the watchdog's state. Using replicated contexts also avoids adding complex synchronization to lock objects during checking. But blindly replicating contexts will incur high overhead. We perform *immutability analysis* [74, 77] on the watchdog contexts. If a context is immutable, OmegaGen generates a reference setter instead, which only holds a reference to the context source.

To further reduce context replication, we use a simple but effective lazy copying approach that, instead of replicating a context upon each set, delays the replication to only when a getter needs it. To deal with potential inconsistency due to lazy replication—*e.g.*, the main program has modified the context after the setter call—we associate a context with several attributes: `version`, `weak_ref` (weak reference to the source object), and `hash` (hash code for the value of the source object). The lazy setter only sets these attributes but does not replicate the context. Later when the getter is invoked, the getter checks if the referent of `weak_ref` is not null. If so, it further checks if the current hash code of the referent's value matches the recorded `hash` and skip replication if they do not match (main program modified context). Besides the attribute checks in getters, the watchdog driver will check if the `version` attributes of each context in a vulnerable operation match and skip the checking if the versions are inconsistent (see further elaboration in Appendix A).

I/O Redirection and Idempotent Wrappers. Besides memory side effects; we also need to prevent I/O side effects. For instance, if a vulnerable operation is writing to a snapshot file, a watchdog could accidentally write to the same snapshot file and affect subsequent executions of the main program. OmegaGen adds I/O redirection capability in watchdogs to address this issue: when OmegaGen generates the context replication code, the replication procedure will check if the context refers to a file-related resource, and if so the context will be replicated with the file path changed to a watchdog test file under the same directory path. Thus watchdogs would experience similar issues such as degraded or faulty storage.

If the storage system being written to is internally load-balanced (*e.g.*, S3), however, the test file may get distributed to a different environment and thus miss issues that only affect the original file. This limitation can be addressed as our write redirection is implemented in a cloning library, so it is relatively easy to extend the logic of deciding the redirection path there to consider the load-balancing policy (if exposed). Besides, if the underlying storage system is layered and complex like S3, it is perhaps better to apply OmegaGen on that system to directly expose partial failures there.

For socket I/O, OmegaGen can perform similar redirection to a special watchdog port if we know beforehand the remote components are also OmegaGen-instrumented. Since this assumption may not hold, OmegaGen by default rewrites the watchdog's socket I/O operation as a ping operation.

If the vulnerable operation is a read-type operation, redirection to read from the watchdog special test file may not help. We design an idempotent wrapper mechanism so that both the main program and watchdog can invoke the wrapper safely. If the main program invokes the wrapper first, it directly performs the actual read-type operation and caches the result in a context. When the watchdog invokes the wrapper, if the main program is in the critical section, it will wait until the main program finishes, and then it gets the cached context. In the normal scenario, the watchdog can use the data from the read operation without performing the actual read. In the faulty scenario, if the main program blocks indefinitely in performing the read-type operation, the watchdog would uncover the hang issue through the timeout of waiting in its wrapper; a bad value from the read would also be captured by the watchdog after retrieving it. For each vulnerable operation of read-type, OmegaGen generates an idempotent wrapper with the above property, replaces the main program's original call instruction to invocation of the wrapper, and places a call instruction to the wrapper in the watchdog checker as well.

5 Implementation

We implemented OmegaGen in Java with 8,100 SLOC. Its core components are built on top of the Soot [90] program analysis framework, so it supports systems in Java bytecode. OmegaGen does not rely on specific JDK features. The Soot version we used can analyze bytecode up to Java 8. We leverage a cloning library [79] with around 400 SLOC of changes to support our selective context replication and I/O redirection mechanisms. OmegaGen's workflow consists of multiple phases to analyze and instrument the program and generate

	ZK	CS	HF	HB	MR	YN
SLOC	28K	102K	219K	728K	191K	229K
Methods	3,562	12,919	79,584	179,821	16,633	10,432

Table 2: Evaluated system software. *ZK*: ZooKeeper; *CS*: Cassandra; *HF*: HDFS; *HB*: HBase; *MR*: MapReduce; *YN*: Yarn.

	ZK	CS	HF	HB	MR	YN
Watchdogs	96	190	174	358	161	88
Methods	118	464	482	795	371	222
Operations	488	2,112	3,416	9,557	6,116	752

Table 3: Number of watchdogs and checkers generated. Not all watchdogs will be activated at runtime.

watchdogs. A single script automates the workflow and packages the watchdogs with the main program into a bundle.

6 Evaluation

We evaluate OmegaGen to answer several questions: (1) does our approach work for large software? (2) can the generated watchdogs detect and localize diverse forms of real-world partial failures? (3) do the watchdogs provide strong isolation? (4) do the watchdogs report false alarms? (5) what is the runtime overhead to the main program? The experiments were performed on a cluster of 10 cloud VMs. Each VM has 4 vCPUs at 2.3GHz, 16 GB memory, and 256 GB disk.

6.1 Generating Watchdogs

To evaluate whether our proposed technique can work for real-world software, we evaluated OmegaGen on six large systems (Table 2). We chose these systems because they are widely used and representative, with codebases as large as 728K SLOC to analyze. OmegaGen uses around 30 lines of default rules for the vulnerable operation heuristics (most are types of Java library methods) and an average of 10 system-specific rules (*e.g.*, special asynchronous wait patterns). OmegaGen successfully generates watchdogs for all six systems.

Table 3 shows the total watchdogs generated. Each watchdog here means a root of reduced methods. Note that these are static watchdogs. Only a subset of them will be activated in production by the watchdog predicates and context hooks (§4.1). We further evaluate how comprehensive the generated checkers are by measuring how many thread classes in the software have at least one watchdog checker generated. Figure 6 shows the results. OmegaGen achieves an average coverage ratio of 60%. For the threads that do not have checkers, they are either not long-running (*e.g.*, auxiliary tools) or OmegaGen did not find vulnerable operations in them. In general, OmegaGen may fail to generate good checkers for modules that primarily perform computations or data structure manipulations. The generated checkers may still contain some redundancy even after the reduction (§4.3).

6.2 Detecting Real-world Partial Failures

Failure Benchmark To evaluate the effectiveness of our generated watchdogs, we collected and reproduced 22 **real-world** partial failures in the six systems. Table 10 in the appendix lists the case links and types. All of these failures led to severe consequences. They involve sophisticated fault injection and workload to trigger. It took us 1 week on average to reproduce each failure. Seven cases are from our study in Section 2. Others are new cases we did not study before.

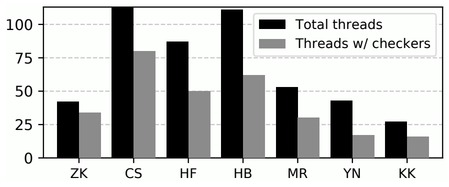


Figure 6: Thread-level coverage by generated watchdog checkers.

Detector	Description
Client (Panorama [75])	instrument and monitor client responses
Probe (Falcon [82])	daemon thread in the process that periodically invokes internal functions with synthetic requests
Signal	script that scans logs and checks JMX [40] metrics
Resource	daemon thread that monitors memory usage, disk and I/O health, and active thread count

Table 4: Four types of baseline detectors we implemented.

Baseline Detectors The built-in detectors (heartbeat) in the six systems cannot handle partial failures at all. We thus implement four types of advanced detectors for comparison (Table 4). The client checker is based on the observers in state-of-the-art work, Panorama [75]. The probe checker presents Falcon [82] app spies (which are also manually written in the Falcon paper). When implementing the signal and resource checkers, we follow the current best practices [15, 42] and monitor signals recommended by practitioners [2, 31, 41, 43].

Methodology The watchdogs and baseline detectors are all configured to run checks *every second*. When reproducing each case, we record when the software reaches the failure program point and when a detector first reports failure. The detection time is the latter minus the former. For slow failures, it is difficult to pick a precise start time. We set the start point using criteria recommended by practitioners, e.g., when number of outstanding requests exceeds 10 for ZooKeeper [31].

Result Table 5 shows the results. Overall, the watchdogs detected 20 out of the 22 cases with a median detection time of 4.2 seconds. 12 of the detected cases are captured by the default vulnerable operation rules. 8 are caught by system-specific rules. In general, the watchdogs were effective for liveness issues like deadlock, indefinite blocking as well as safety issues that trigger explicit error signals or exceptions. But they are less effective for silent correctness errors.

In comparison, as Table 5 shows, the best baseline detector only detected 11 cases. Even the combination of all baseline detectors detected only 14 cases. The client checkers missed 68% of the failures because these failures concern the internal functionality or some optimizations that are not immediately visible to clients. The signal checker is the most effective among the baseline detectors, but it is also noisy (§6.6).

Case Studies ZK1 [45]: This is the running example in

	ZK1	ZK2	ZK3	ZK4	CS1	CS2	CS3	CS4	HF1	HF2	HF3	HF4	HB1	HB2	HB3	HB4	HB5	MR1	MR2	MR3	MR4	YN1
Watch.	4.28	-5.89	3.00	41.19	-3.73	4.63	46.56	38.72	1.10	6.20	3.17	2.11	5.41	7.89	✖	0.80	5.89	1.01	4.07	1.46	4.68	✖
Client	✖	2.47	2.27	✖	441	✖	✖	✖	✖	✖	✖	✖	✖	4.81	✖	6.62	✖	✖	✖	✖	8.54	7.38
Probe	✖	✖	✖	✖	15.84	✖	✖	✖	✖	✖	✖	✖	✖	4.71	✖	7.76	✖	✖	✖	✖	✖	✖
Signal	12.2	0.63	1.59	0.4	5.31	✖	✖	✖	✖	✖	✖	0.77	0.619	✖	0.62	61.0	✖	✖	✖	✖	0.60	1.16
Res.	5.33	0.56	0.72	17.17	209.5	✖	-19.65	✖	-3.13	✖	✖	0.83	✖	✖	✖	0.60	✖	✖	✖	✖	✖	✖

Table 5: Detection times (in seconds) for the real-world cases in Table 10. ✖: undetected.

	ZK1	ZK2	ZK3	ZK4	CS1	CS2	CS3	CS4	HF1	HF2	HF3	HF4	HB1	HB2	HB3	HB4	HB5	MR1	MR2	MR3	MR4	YN1
Watchdog	➻	➻	●	✻	➻	✻	●	✻	✻	✺	➻	➻	➻	➻	n/a	➻	✺	➻	➻	✺	➻	n/a
Client	n/a	●	●	n/a	●	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	●	n/a	❍	n/a	n/a	n/a	n/a	●	●
Probe	n/a	n/a	n/a	n/a	◗	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	◗	n/a	◗	n/a	n/a	n/a	n/a	n/a	n/a
Signal	●	➻	●	●	➻	n/a	n/a	n/a	n/a	n/a	n/a	➻	➻	n/a	✺	✺	n/a	n/a	n/a	n/a	➻	➻
Resource	●	●	●	●	●	n/a	●	n/a	●	n/a	n/a	●	n/a	n/a	n/a	●	n/a	n/a	n/a	n/a	n/a	n/a

Table 6: Failure localization for the real-world cases in Table 10.➻: pinpoint the faulty instr. ✻: pinpoint the faulty function or data structure. ✺: pinpoint a func in the faulty function's call chain. ◗: pinpoint some entry function in the program, which is distant from the root cause. ●: only pinpoint the faulty process. ❍: misleadingly pinpoint another innocent process. n/a: not applicable because failure is undetected.

the paper. A network issue caused a ZooKeeper remote snapshot dumping operation to be blocked in a critical section, which prevented update-type request processing threads from proceeding (Figure 1). OmegaGen generates a checker `serializeNode_reduced`, which exposed the issue in 4 s.

CS1 [7]: The Cassandra Commitlog executor accidentally died due to a bad commit disk volume. This caused the uncommitted writes to pile up, which in turn led to extensive garbage collection and the process entering a zombie status. The relevant watchdog OmegaGen generates is `CommitLogSegment_reduced`. Interestingly, this case had negative detection time. This happens because the executor successfully executed the faulty program point prior to the failure and set the watchdog context (log segment path). When the checker was scheduled, the context was still valid so the checker was activated and exposed the issue ahead of time.

HB5 [18]: Users observed some gigantic write-ahead-logs (WALs) on their HBase cluster even when WAL rolling is enabled. This is because when a peer is previously removed, one thread gets blocked for sending a shutdown request to a closed executor. Unfortunately this procedure holds the same lock `ReplicationSourceManager#recordLog`, which does the WAL rolling (to truncate logs). Our generated watchdog mimics the procedure of submitting request and waiting for completion, and experienced the same stalling issue on closed executor.

CS4 [11]: Due to a severe performance bug in the Cassandra compaction module, all the `RangeTombstones` ever created for the partition that have expired would remain in memory until the compaction completes. The compaction task would be very slow when the workloads contain a lot of overwrites to collections. The relevant checker OmegaGen generates is `SSTableWriter#append_reduced`. After the tombstones piles up, this checker reports a slow alert based on the dramatic ($10\times$) increase of moving average of operation latencies.

YN1 [44]: A new application (AM) was stuck after getting allocated to a recently added NodeManager (NM). This was caused by `/etc/hosts` on the ResourceManager (RM) not being updated, so this new NM was unresolvable when RM built the service tokens. RM would retry forever and the AM would keep getting allocated to the same NM. Our watchdogs failed to detect the issue. The reason is that the faulty operation `buildTokenService()` mainly creates some data structure, so OmegaGen failed to consider it as vulnerable.

6.3 Localizing Partial Failure

Detection is only the first step. We further evaluate the localization effectiveness for the detected cases in Table 5. we measure the distance between the error reporting location and the faulty program point. We categorize the distance into six levels of decreasing accuracy. Table 6 shows the result. Watchdogs directly pinpoint the faulty instruction for 55% (11/20) of the detected cases, which indicates the effectiveness of our vulnerable operation heuristics. In case MR1 [35], after noticing the symptom (reducer did not make progress for a long time), it took the user more than two days of careful log analysis and thread dumps to narrow down the cause. With the watchdog error report, the fault was obvious.

For 35% (7/20) of detected cases, the watchdogs either localize to some program point within the same function or some function along the call chain, which can still significantly ease troubleshooting. For example, in case HF2 [24], the balancer was stuck in a loop in `waitForMoveCompletion()` because `isPendingQEmpty()` will return false when no mover threads are available. The generated watchdog did not pinpoint either place. But it caught the error through timeout in executing a `future.get()` vulnerable operation in its checker `dispatchBlockMoves_reduced`, which narrows down the issue.

In comparison, the client or resource detectors can only pinpoint the faulty process. To narrow down the fault, users must spend significant time analyzing logs and code. In case HB4 [21], the client checker even blamed a wrong innocent process, which would completely mislead the diagnosis. The probe checker localizes failures to some internal functions in the program. But these functions are still too high-level and distant from the fault. The signal checker localizes 8 cases.

6.4 Fault-Injection Tests

To evaluate how the watchdogs may perform in real deployment, we conducted a random fault-injection experiment on

	ZK	CS	HF	HB	MR	YN
watch.	0–0.73	0–1.2	0	0–0.39	0	0–0.31
watch_v.	0–0.01	0	0	0–0.07	0	0
probe	0	0	0	0	0	0
resource	0–3.4	0–6.3	0.05–3.5	0–3.72	0.33–0.67	0–6.1
signal	3.2–9.6	0	0–0.05	0–0.67	0	0

Table 7: False alarm ratios (%) of all detectors in the evaluated six systems. Each cell reports the ratio range under three setups (stable, loaded, tolerable). *watch_v*: watchdog with validators.

the latest ZooKeeper. In particular, we inject four types of faults to the system: *Infinite loop* (modify loop condition to force running forever); *Arbitrary delay* (inject 30 seconds delay in some complex operations); *System resource contention* (exhaust CPU/memory resource); *I/O delay* (inject 30 seconds delay in file system or network). After that, we run a series of workloads and operations (e.g., restart some server). We successfully trigger 16 synthetic failures. Our generated watchdogs can detect 13 out of the 16 triggered synthetic failures with a median detection time of 6.1 seconds. The watchdogs pinpoint the injected failure scope for 11 cases.

6.5 Discovering A New Partial Failure Bug

During our continuous testing, our watchdogs exposed a new partial bug in the latest version (3.5.5) of ZooKeeper. We observe that our ZooKeeper cluster occasionally hangs and new create requests time out while the admin tool still shows the leader process is working. This symptom is similar to our studied bug ZK1. But that bug is already fixed in the latest version. The issue is also non-deterministic. Our watchdogs report the failure in 4.7 seconds. The watchdog log helps us pinpoint the root cause for this puzzling failure. The log shows the checker that reported the issue was `serializeAcls_reduced`. We further inspected this function and found that the problem was the server serializing the `ACLCache` inside a critical section. When developers fixed the ZK1 bug, this similar flaw was overlooked and recent refactoring of this class made the flaw more problematic. We reported this new bug [49], which has been confirmed by the developers and fixed.

6.6 Side Effects and False Alarms

We ran the watchdog-enhanced systems with extensive workloads and verified that the systems pass their own tests. We also verified the integrity of the files and client responses by comparing them with ones from the vanilla systems. If we disable our side-effect prevention mechanisms (§4.7), however, the systems would experience noticeable anomalies, *e.g.*, snapshots get corrupted, system crash; or, the main program would hang because the watchdog read the data from a stream.

We further evaluate the false alarms of watchdogs and baseline detectors under three setups: *stable*: runs fault-free for 12 hours with moderate workloads (§6.7); *loaded*: random node restarts, every 3 minutes into the moderate workloads, switch to aggressive workloads (3× number of clients and 5× request sizes); *tolerable*: run with injected transient errors tolerable by the system. Table 7 shows the results. The false alarm ratio is calculated from total false failure reports divided by the total number of check executions. Watchdogs did *not* report false alarms in the stable setup. But during a loaded period, they incur around 1% false alarms due to socket connection errors or resource contention. These false alarms would disappear once the transient faults are gone. With the validator mechanism (§4.6), the watchdog false alarm ratios (the *watch_v* row) are significantly reduced. Among the baseline detectors, we can see that even though signal checkers achieved better detection, they incur high false alarms (3–10%).

	ZK	CS	HF	HB	MR	YN
Analysis	21	166	75	92	55	50
Generation	43	103	130	953	131	89

Table 8: OmegaGen watchdog generation time (sec).

	ZK	CS	HF	HB	MR	YN
Base	428.0	3174.9	90.6	387.1	45.0	45.0
w/ Watch.	399.8	3014.7	85.1	366.4	42.1	42.3
w/ Probe.	417.6	3128.2	89.4	374.3	44.9	44.9
w/ Resource.	424.8	3145.4	89.9	385.6	44.9	44.6

Table 9: System throughput (op/s) w/ different detectors.

6.7 Performance and Overhead

We first measure the performance of OmegaGen's static analysis. Table 8 shows the results. For all but HBase, the whole process takes less than 5 minutes. HBase takes 17 minutes to generate watchdogs because of its large codebase.

We next measure the runtime overhead of enabling watchdogs and the baseline detectors. We used popular benchmarks configured as follows: for ZK, we used an open-source benchmark [16] with 15 clients sending 15,000 requests (40% read); for Cassandra, we used YCSB [61] with 40 clients sending 100,000 requests (50% read); for HDFS, we used built-in benchmark NNBenchWithoutMR which creates and writes 100 files, each file has 160 blocks and each block is 1MB; for HBase, we used YCSB with 40 clients sending 50,000 requests (50% read); for MapReduce and Yarn, we used built-in DFSIO benchmark which writes 400 10MB files.

Table 9 shows that the watchdogs incur 5.0%–6.6% overhead on throughput. The main overhead comes from the watchdog hooks rather than the concurrent checker execution. The probe detectors are more lightweight, incurring 0.2%–3.2% overhead. We also measure the latency impact. The watchdogs incur 9.3%–12.2% overhead on average latency and 8.3%–14.0% overhead on tail (99th percentile) latency. But given the watchdog's significant advantage in failure detection and localization, we believe its higher overhead is justified. For a cloud infrastructure, operators could also choose to activate watchdogs on a subset of the deployed nodes to reduce the overhead while still achieving good coverage.

We measure the CPU usages of each system w/o and w/ watchdogs. The results are 57%→66% (ZK), 199%→212% (CS), 33%→38% (HF), 36%→41% (HB), 5.6%→6.9% (MR), 1.5%→3% (YN). We also analyze the heap memory usages. The median memory usages (in MiB) are 128→131 (ZK), 447→459 (CS), 165→178 (HF), 197→201 (HB), 152→166

(MR), 154→157 (YN). The increase is small because contexts are only lazily replicated every checking interval, compared to continuous object allocations in the main program.

6.8 Sensitivity

We evaluate the sensitivity of our default 4-sec timeout threshold on detecting liveness issues with ZK1 [45] (stuck failure) and ZK4 [48] (slow failure). Under timeout threshold 100 ms, 300 ms, 500 ms, 1 s, 4 s, and 10 s, the detection times for ZK1 are respectively 0.51 s, 0.61 s, 0.70 s, 1.32 s, 4.28 s, and 12.09 s. The detection time generally decreases with smaller timeout, but it is bounded by the checking interval. With timeout of 100 ms, we observe 6 false positives in 5 minutes. For ZK4, when the timeout threshold is aggressive, the slow fault can be detected without the moving average mechanism (§4.5), in particular with detection times of 61.65 s (100 ms), 91.38 s (300 ms), 110.32 s (500 ms). Eventually the resource leak exhausts all available memory before the watchdog exceeds more conservative thresholds.

7 Limitations

OmegaGen has several limitations we plan to address in future work: (1) Our vulnerable operation analysis is heuristics-based. This step can be improved through offline profiling or dynamic adaptive selection. (2) Our generated watchdogs are effective for liveness issues and common safety violations. But they are ineffective to catch silent semantic failures. We plan to leverage existing resources that contain semantic hints such as test cases to derive runtime semantic checks. (3) OmegaGen achieves memory isolation with static analysis-assisted context replication. We will explore more efficient solutions like copy-on-write when porting OmegaGen to C/C++ systems. (4) OmegaGen generates watchdogs to report failures for individual process. One improvement is to pair OmegaGen with failure detector overlays [89] so the failure detector of one process could inspect another process' watchdogs. (5) Our watchdogs currently focus on fault detection and localization but not recovery. We will integrate microreboot [58] and ROC techniques [87].

8 Related Work

Partial failure has been discussed in multiple contexts. Arpaci-Dusseau and Arpaci-Dusseau propose the fail-stutter fault model [56]. Prabhakaran *et al.* analyze the fail-partial model for disks [88]. Correia *et al.* propose the ASC fault model [62]. Huang *et al.* propose a definition for gray failure in cloud [76]. Gunawi *et al.* [68] studies the fail-slow performance faults in hardware. Our study presented in Section 2 focuses on partial failures in modern cloud software. A recent work analyzes failures in cloud systems caused by network partitions [54]. Our work's scope is at the process granularity. A network partition may causes total failures to the partitioned processes (disconnected from other processes). Besides, our work covers much more diverse root causes beyond network issues.

Failure detection has been extensively studied [53, 59, 60, 63, 65, 71, 72, 80–82, 91]. But they primarily focus on detecting fail-stop failures in distributed systems; partial failures are beyond the scope of these detectors. Panorama [75] proposes to leverage observability in a system to detect gray failures [76]. While this approach can enhance failure detection, it assumes some external components happen to observe the subtle failure behavior. These logical observers also cannot isolate which part of the failing process is problematic, making subsequent failure diagnosis time-consuming [32].

Watchdog timers are essential hardware components found in embedded systems [57]. For general-purpose software, watchdogs are more challenging to construct manually due to the large size of the codebase and complex program logic. Consequently, existing software using the watchdog concept [4, 14] only designs watchdogs as shallow health checks (*e.g.*, http test) and a kill policy [42]. Our position paper [83] advocates for the intrinsic watchdog abstraction and articulates its design principles. OmegaGen provides the ability to automatically generate comprehensive, customized watchdogs for a given program through static analysis.

Several works aim to generate software invariants or ease runtime checking. Daikon [64] infers likely program invariants from dynamic execution traces. PCHECK [92] uses program slicing to extract configuration checks to detect latent misconfiguration during initialization. OmegaGen is complementary to these efforts. We focus on synthesizing checkers for monitoring long-running procedures of a program in production by using a novel program reduction technique.

9 Conclusion

System software continues to become ever more complex. This leads to a variety of partial failures that are not captured by existing solutions. This work first presents a study of 100 real-world partial failures in popular system software to shed light on the characteristics of such failures. We then present OmegaGen, which takes a program reduction approach to generate watchdogs for detecting and localizing partial failures. Evaluating OmegaGen on six large systems, it can generate tens to hundreds of customized watchdogs for each system. The generated watchdogs detect 20 out of 22 real-world partial failures with a median detection time of 4.2 seconds, and pinpoint the scope of failure for 18 cases; these results significantly outperform the baseline detectors. Our watchdogs also exposed a new partial failure in latest ZooKeeper.

Acknowledgments

We would like to thank the NSDI reviewers and our shepherd, Aurojit Panda, for their valuable comments that improved the paper. We thank Ziyan Wang for implementing the `wd_assert` API support for OmegaGen. We are grateful to the generous cloud research credits support from Azure, AWS and Google Cloud Platform. This work was supported in part by funding from NSF grants CNS-1755737 and CNS-1910133.

References

[1] Alibaba cloud reports IO hang error in north China. https://equalocean.com/technology/20190303-alibaba-cloud-reports-io-hang-error-in-north-china.

[2] Apache Cassandra: Some useful JMX metrics to monitor. https://medium.com/@foundev/apache-cassandra-some-useful-jmx-metrics-to-monitor-7f1d3ede294a.

[3] Apache module mod_proxy_hcheck. https://httpd.apache.org/docs/2.4/mod/mod_proxy_hcheck.html.

[4] Apache module mod_watchdog. https://httpd.apache.org/docs/2.4/mod/mod_watchdog.html.

[5] Cassandra-10477: java.lang.AssertionError in StorageProxy.submitHint. https://issues.apache.org/jira/browse/CASSANDRA-10477.

[6] Cassandra-5229: streaming tasks hang in netstats. https://issues.apache.org/jira/browse/CASSANDRA-5229.

[7] Cassandra-6364: Commit log executor dies and causes unflushed writes to quickly accumulate. https://issues.apache.org/jira/browse/CASSANDRA-6364.

[8] Cassandra-6415: Snapshot repair blocks forever if something happens to the remote response. https://issues.apache.org/jira/browse/CASSANDRA-6415.

[9] Cassandra-6788: Race condition silently kills thrift server. https://issues.apache.org/jira/browse/CASSANDRA-6788.

[10] Cassandra-8447: Nodes stuck in CMS GC cycle with very little traffic when compaction is enabled. https://issues.apache.org/jira/browse/CASSANDRA-8447.

[11] Cassandra-9486: LazilyCompactedRow accumulates all expired RangeTombstones. https://issues.apache.org/jira/browse/CASSANDRA-9486.

[12] Cassandra-9549: Memory leak in Ref.GlobalState due to pathological ConcurrentLinkedQueue.remove behaviour. https://issues.apache.org/jira/browse/CASSANDRA-9549.

[13] Cassandra: demystify failure detector, consider partial failure handling, latency optimizations. https://issues.apache.org/jira/browse/CASSANDRA-3927.

[14] Cloud computing patterns: Watchdog. http://www.cloudcomputingpatterns.org/watchdog/.

[15] Consul health check. https://www.consul.io/docs/agent/checks.html.

[16] Distributed database benchmark tester. https://github.com/etcd-io/dbtester.

[17] GoCardless service outage on October 10th, 2017. https://gocardless.com/blog/incident-review-api-and-dashboard-outage-on-10th-october.

[18] HBASE-16081: Removing peer in replication not gracefully finishing blocks WAL rolling. https://issues.apache.org/jira/browse/HBASE-16081.

[19] HBASE-16429: FSHLog deadlock if rollWriter called when ring buffer filled with appends. https://issues.apache.org/jira/browse/HBASE-16429.

[20] HBASE-18137: Empty WALs cause replication queue to get stuck. https://issues.apache.org/jira/browse/HBASE-18137.

[21] HBASE-21357: Reader thread encounters out of memory error. https://issues.apache.org/jira/browse/HBASE-21357.

[22] HBASE-21464: Splitting blocked with meta NSRE during split transaction. https://issues.apache.org/jira/browse/HBASE-21464.

[23] HDFS-11352: Potential deadlock in NN when failing over. https://issues.apache.org/jira/browse/HDFS-11352.

[24] HDFS-11377: Balancer hung due to no available mover threads. https://issues.apache.org/jira/browse/HDFS-11377.

[25] HDFS-12070: Failed block recovery leaves files open indefinitely and at risk for data loss. https://issues.apache.org/jira/browse/HDFS-12070.

[26] HDFS-2882: DN continues to start up, even if block pool fails to initialize. https://issues.apache.org/jira/browse/HDFS-2882.

[27] HDFS-4176: EditLogTailer should call rollEdits with a timeout. https://issues.apache.org/jira/browse/HDFS-4176.

[28] HDFS-4233: NN keeps serving even after no journals started while rolling edit. https://issues.apache.org/jira/browse/HDFS-4233.

[29] HDFS-8429: Error in DomainSocketWatcher causes others threads to be stuck threads. https://issues.apache.org/jira/browse/HDFS-8429.

[30] HDFS short-circuit local reads. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.

[31] How to monitor Zookeeper. https://blog.serverdensity.com/how-to-monitor-zookeeper/.

[32] Just say no to more end-to-end tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html.

[33] MapReduce-3634: Dispatchers in daemons get exceptions and silently stop processing. https://issues.apache.org/jira/browse/MAPREDUCE-3634.

[34] MapReduce-6190: Job stuck for hours because one of the mappers never started up fully. https://issues.apache.org/jira/browse/MAPREDUCE-6190.

[35] MapReduce-6351: Circular wait in handling errors causes reducer to hang in copy phase. https://issues.apache.org/jira/browse/MAPREDUCE-6351.

[36] MapReduce-6957: Shuffle hangs after a node manager connection timeout. https://issues.apache.org/jira/browse/MAPREDUCE-6957.

[37] Mesos-8830: Agent gc on old slave sandboxes could empty persistent volume data. https://issues.apache.org/jira/browse/MESOS-8830.

[38] mod_proxy_ajp: mixed up response after client connection abort. https://bz.apache.org/bugzilla/show_bug.cgi?id=53727.

[39] Office 365 update on recent customer issues. https://blogs.office.com/2012/11/13/update-on-recent-customer-issues/.

[40] Overview of the JMX technology. https://docs.oracle.com/javase/tutorial/jmx/overview/index.html.

[41] Running ZooKeeper in production. https://docs.confluent.io/current/zookeeper/deployment.html.

[42] Task health checking and generalized checks. http://mesos.apache.org/documentation/latest/health-checks.

[43] Tuning a database cluster with the performance service. https://docs.datastax.com/en/opscenter/6.1/opsc/online_help/services/tuneClusterPerfService.html.

[44] Yarn-4254: Accepting unresolvable NM into cluster causes RM to retry forever. https://issues.apache.org/jira/browse/YARN-4254.

[45] ZooKeeper-2201: Network issue causes cluster to hang due to blocking I/O in synch. https://issues.apache.org/jira/browse/ZOOKEEPER-2201.

[46] ZooKeeper-2319: UnresolvedAddressException cause the listener exit. https://issues.apache.org/jira/browse/ZOOKEEPER-2319.

[47] ZooKeeper-2325: Data inconsistency when all snapshots empty or missing. https://issues.apache.org/jira/browse/ZOOKEEPER-2325.

[48] ZooKeeper-3131: WatchManager resource leak. https://issues.apache.org/jira/browse/ZOOKEEPER-3131.

[49] ZooKeeper-3531: Synchronization on ACLCache cause cluster to hang. https://issues.apache.org/jira/browse/ZOOKEEPER-3531.

[50] ZooKeeper-914: QuorumCnxManager blocks forever. https://issues.apache.org/jira/browse/ZOOKEEPER-914.

[51] Twilio billing incident post-mortem: Breakdown, analysis and root cause. https://bit.ly/2V8rurP, July 23, 2013.

[52] Google compute engine incident 17008. https://status.cloud.google.com/incident/compute/17008, June 17, 2017.

[53] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS '09, pages 3–3, Monte Verità, Switzerland, 2009.

[54] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 51–68, Carlsbad, CA, USA, 2018.

[55] Amazon. AWS service outage on October 22nd, 2012. https://aws.amazon.com/message/680342.

[56] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HotOS '01, pages 33–. IEEE Computer Society, 2001.

[57] A. S. Berger. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. CMP Books. Taylor & Francis, 2001.

[58] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI '04, pages 31–44, San Francisco, CA, 2004.

[59] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[60] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, May 2002.

[61] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, Indianapolis, Indiana, USA, 2010.

[62] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 41–41, Boston, MA, 2012.

[63] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 303–312. IEEE Computer Society, 2002.

[64] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, Los Angeles, California, USA, 1999.

[65] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, Feb. 2003.

[66] E. Gilman. The discovery of Apache ZooKeeper's poison packet. https://www.pagerduty.com/blog/the-discovery-of-apache-zookeepers-poison-packet, May 7, 2015.

[67] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 14:1–14:16, San Jose, California, 2008.

[68] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Oakland, CA, USA, 2018.

[69] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM SIGCOMM Conference*, SIGCOMM '15, pages 139–152, London, United Kingdom, 2015.

[70] A. Gupta and J. Shute. High-Availability at massive scale: Building Google's data infrastructure for Ads. In *Proceedings of the 9th International Workshop on Business Intelligence for the Real Time Enterprise*, BIRTE '15, 2015.

[71] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 175–188, Stevenson, Washington, USA, 2007.

[72] A. Haeberlen and P. Kuznetsov. The fault detection problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 99–114, Nîmes, France, 2009.

[73] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The φ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 66–78, Florianópolis, Brazil, 2004.

[74] B. Holland, G. R. Santhanam, and S. Kothari. Transferring state-of-the-art immutability analyses: Experimentation toolbox and accuracy benchmark. In *IEEE International Conference on Software Testing, Verification and Validation*, ICST '17, pages 484–491, March 2017.

[75] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[76] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI. ACM, May 2017.

[77] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. Reim & ReImInfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 879–896, Tucson, Arizona, USA, 2012.

[78] D. King. Partial Failures are Worse Than Total Failures. https://www.tildedave.com/2014/03/01/application-failure-scenarios-with-cassandra.html, March 2014.

[79] K. Kougios. Java cloning library. https://github.com/kostaskougios/cloning.

[80] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 427–442, Lombard, IL, Apr. 2013.

[81] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, Bordeaux, France, 2015.

[82] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, Cascais, Portugal, Oct. 2011.

[83] C. Lou, P. Huang, and S. Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 51–57, Bertinoro, Italy, 2019.

[84] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors – a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb 1988.

[85] Microsoft. Details of the December 28th, 2012 Windows Azure storage disruption in US south. `https://bit.ly/2Iofhcz`, January 16, 2013.

[86] D. Nadolny. Debugging distributed systems. In *SREcon 2016*, April 7-8 2016.

[87] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, and et al. Recovery oriented computing (ROC): Motivation, definition, techniques,. Technical report, USA, 2002.

[88] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 206–220, Brighton, United Kingdom, 2005.

[89] L. Suresh, D. Malkhi, P. Gopalan, I. P. Carreiro, and Z. Lokhandwala. Stable and consistent membership at scale with Rapid. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18, page 387–399, Boston, MA, USA, 2018.

[90] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[91] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 55–70, The Lake District, United Kingdom, 1998.

[92] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early detection of configuration errors to reduce failure damage. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 619–634, Savannah, GA, USA, 2016.

[93] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, Broomfield, CO, 2014.

Appendix A Additional Clarifications

Consistency under Lazy Replication Section 4.7 describes that we associate a context with three attributes (`version`, `weak_ref`, and `hash`) to deal with potential inconsistency due to the lazy replication optimization. Here, we give a concrete example to clarify how potential inconsistency could arise and how it is addressed. With lazy replication (essentially "copy-on-get"), a context may be modified or even invalidated after the context setter call; if this occurs, the getter will replicate a different context value. For example,

```
       Main Program                    Watchdog Checker
-----------------------------    --------------------------
void foo() {                     void foo_reduced_invoke() {
  foo_reduced_args_setter(oa);
  write(oa);
  oa.append("test");
                <---      oa = foo_reduced_ctx.args_getter(0);
}
```

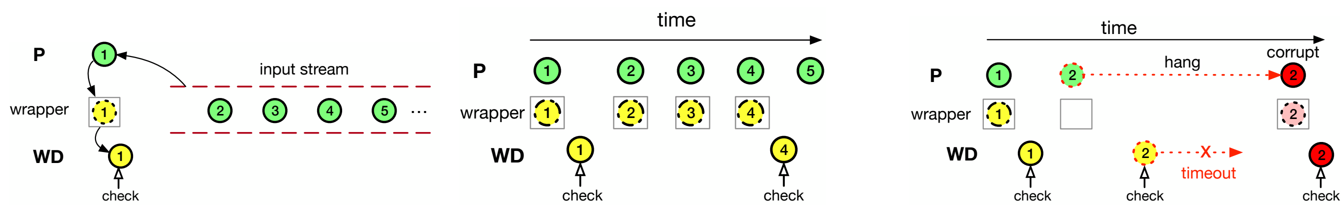
Id.	Root Cause	Conseq.	Sticky?	Study?
ZK1 [45]	Bad Synch.	Stuck	No	Yes
ZK2 [66]	Uncaught Error	Zombie	Yes	Yes
ZK3 [47]	Logic Error	Inconsist.	Yes	No
ZK4 [48]	Resource Leak	Slow	Yes	Yes
CS1 [7]	Uncaught Error	Zombie	Yes	Yes
CS2 [8]	Indefinite Blocking	Stuck	No	Yes
CS3 [12]	Resource Leak	Slow	Yes	No
CS4 [11]	Performance Bug	Slow	Yes	No
HF1 [29]	Uncaught Error	Stuck	Yes	Yes
HF2 [24]	Indefinite Blocking	Stuck	No	Yes
HF3 [23]	Deadlock	Stuck	Yes	No
HF4 [28]	Uncaught Error	Data Loss	Yes	No
HB1 [20]	Infinite Loop	Stuck	Yes	No
HB2 [19]	Deadlock	Stuck	Yes	No
HB3 [22]	Logic Error	Stuck	Yes	No
HB4 [21]	Uncaught Error	Denial	Yes	No
HB5 [18]	Indefinite Blocking	Silent	Yes	No
MR1 [35]	Deadlock	Stuck	Yes	No
MR2 [34]	Infinite Loop	Stuck	Yes	No
MR3 [36]	Improper Err Handling	Stuck	Yes	No
MR4 [33]	Uncaught Error	Zombie	Yes	No
YN1 [44]	Improper Err Handling	Stuck	Yes	No

Table 10: 22 real-world partial failures reproduced for evaluation. *ZK*: ZooKeeper; *CS*: Cassandra; *HF*: HDFS; *HB*: HBase; *MR*: MapReduce; *YN*: Yarn. Sticky?: whether the failure persists forever. Study?: whether the failure is from the studied cases in Section 2.

By the time the context getter is invoked in the checker, `oa` may already be invalidated (garbage collected). But since the getter will check the `weak_ref` attribute, it will find out the fact that the context is invalid (`weak_ref` returns `null`) and hence not replicate. If `oa` is still valid, the context getter will further check the hash code of the current value and skip replication if it does not match the recorded `hash`. This approach is lightweight. But it assumes the hash code contract of Java objects being honored in a program. If this is not the case, *e.g.*, `oa`'s hash code is the same regardless of its content, inconsistency (getter replicates a modified context) could arise. Such inconsistency may or may not cause an issue for the checker. For the above example, the checker's `write` may write `"xxxtest"` instead of `"xxx"` to the watchdog test file, which is still fine. But if another vulnerable operation has a special invariant on `"xxx"`, the inconsistency will lead to a false alarm at runtime. Our low false alarm rates during the 12-hour experiment period suggest that hash code contract violation is generally not a major concern for mature software.

Another consistency scenario to consider is when a checker uses some vulnerable operation that requires multiple context arguments. Since the context retrieval is asynchronous under the lazy replication optimization, a race condition could occur while a getter is retrieving all the arguments. For example,

```
           Main Program                      Watchdog Checker
-----------------------------------   --------------------------
```



(a) Idempotent wrapper for `readRecord` **(b)** Invoking wrapper in normal scenario **(c)** Invoking wrapper in faulty scenario

Figure 7: Illustration of idempotent wrapper

```
// called in a loop
synchronized void foo() {              void foo_reduced_invoke() {
                   <--- arg0 = foo_reduced_ctx.args_getter(0);
  ...
  foo_reduced_args_setter(oa, node);
  oa.writeRecord(node);
                   <---  arg1 = foo_reduced_ctx.args_getter(1);
}
```

After the getter retrieves `oa`, the second argument (`node`) is updated before the getter retrieves it. In this case, both arguments are valid and match their recorded `hash` attributes. However, they are mixed from two invocations of `foo()`. We address this inconsistency scenario with the `version` attributes. A checker will compare if the `version` attributes of all the contexts it needs are the same before invoking the checked operation, and skip the checking if the versions are inconsistent.

Appendix B Implementation Details

Idempotent Wrapper Section 4.7 describes our idempotent wrapper mechanism that allows watchdogs to safely invoke non-idempotent operations, especially `read`-type operations. We further elaborate the details for this mechanism here.

The basic idea is to have both the watchdog and main program invoke the wrapper instead of the original operation in a coordinated fashion. The wrapper distinguishes whether the call is from main program or the watchdog. Take a vulnerable operation `readRecord` as an example. In the fault-free scenario, the main program performs the actual `readRecord` like normal; the watchdog checker would get a cached value. In a faulty scenario, the main program may get stuck in `readRecord`; the watchdog would be blocked outside the critical section of the wrapper so it can detect the hang without performing the actual `readRecord`. Figure 7 illustrates both scenarios.

OmegaGen automatically generates idempotent wrappers for all read-type vulnerable operations. OmegaGen first locates all statements that invoke a read operation in the main program. It extracts the stream objects from these statements. A wrapper is generated for each type of stream object. The watchdog driver maintains a map between the stream objects and the wrapper instances. For the wrapper to later perform the actual operation, OmegaGen assigns a distinct operation number for each read-type method in the stream class, and generates a dispatcher that calls the method based on the op number. Then, OmegaGen replaces the original invocation with a call to the watchdog driver's wrapper entry point using the stream object, operation number, and caller source as the arguments. For example, `buf = istream.read();` in the main program would be replaced with `buf = WatchdogDriver.readHelp(istream, 1, 0);` where 1 is the op number for `read` and 0 means the wrapper is called from the main program.

The other steps in the checker construction for the read-type operations are similar to other types of vulnerable operations. The key difference is that OmegaGen will generate a self-contained checker for the *wrapped* operation instead of the operation. It particular, the checker OmegaGen generates will contain a call instruction to the proper wrapper using source 1 (from watchdog) as the argument.

		ZK	CS	HF	HB	MR	YN
Disk	Base	3.97	6.04	88.26	1.50	0.10	0.05
(MB/s)	w/ WD	4.04	6.12	89.02	1.53	0.10	0.05
Network	Base	997	2,884	27	993	1.3	1.5
(KB/s)	w/ WD	1,031	2,915	28	1,048	1.7	1.8

Table 11: Average disk and network I/O usages of the base systems and w/ watchdogs.

Appendix C Supplementary Evaluation

Semantic Check API Our experiments in Section 6 did not use semantic checks, `wd_assert` (§4.5), to avoid biased results. But we did test using `wd_assert` on a hard case ZK3. Although the watchdog OmegaGen automatically generates detected this case, it is because the failure-triggering condition (bad disk) also affected some other vulnerable I/O operations in the watchdog. We wrote a `wd_assert` to check if the on-disk transaction records are far behind in-memory records:

```
wd_assert(lastProcessedZxid <= (new
ZKDatabase(txnLogFactory)).loadDataBase()+MISS_TXN_THRESHOLD);
```

OmegaGen handles the tedious details by automatically extracting the necessary context, encapsulating a watchdog checker, and removing this expensive statement from the main program. The resulted semantic checker can detect the failure within 2 seconds and pinpoint the issue.

I/O Usage Overhead We measured the disk I/O usages (using `iotop`) and network I/O usage (using `nethogs`) for the six systems with and without watchdogs under the same setup as our overhead experiment in Section 6.7. Table 11 shows the results. We can see the I/O usage increase incurred by the watchdogs is small (a median of 1.6% for disk I/O and 4.4% for network I/O).