北京理工大學

本科生毕业设计（论文）外文翻译

外文原文题目：**Understanding, Detecting and Localizing Partial Failures in Large System Software**

中文翻译题目：大型系统软件部分故障的理解、检测与定位

北京理工大学本科生毕业设计（论文）题目

The Subject of Undergraduate Graduation Project (Thesis) of Beijing Institute of Technology

学　　院：	计算机学院
专　　业：	计算机科学与技术（二学位）
学生姓名：	李 雯
学　　号：	1120205029
指导教师：	陆慧梅

大型系统软件 部分故障的理解、检测与定位

本文收录在第 17 届 USENIX 网络系统设计与实施研讨会论文集 (NSDI ’ 20) 中（2020 年 2 月 25 日至 27 日 • 美国加利福尼亚州圣克拉拉）

开源获取会议记录：第十七届 USENIX 网络研讨会系统设计与实施（NSDI ’20）（NetApp 赞助）

作者：Chang Lou ，Peng Huang，Scott Smith（约翰霍普金斯大学）

摘要

云系统中经常发生部分故障，并可能产生严重的危害，包括不一致和数据丢失。不幸的是，这些故障并没有被很好地理解，也不能进行有效地检测。在本文中，我们首先研究了来自五个成熟系统的 100 个真实部分故障，以理解它们的特征。我们发现这些故障是由触发生产环境特有条件的各种缺陷引起的。手动编写有效的检测器来系统地检测此类故障既耗时又容易出错。因此，我们提出了 OmegaGen，这是一种静态分析工具，它通过使用新颖的程序缩减技术为给定程序自动生成定制的看门狗。我们已经成功地将 OmegaGen 应用于六个大型分布式系统。在评估这些系统中的 22 个实际部分故障案例时，生成的看门狗可以检测 20 个案例，中位检测时间为 4.2 秒，并确定 18 个案例的故障范围。生成的看门狗还在最新版本的 ZooKeeper 中揭秘了一个未知的、常见的部分故障错误。

1. 引言

构建永不失误的大型软件是难以把握的。因此，健壮系统的设计人员必须设计运行机制，以主动检查程序是否仍在正常运行，如果没有则作出回应。许多这些机制都是基于一个简单的假设，即程序会由于死机、中断或网络断开而完全停止。

然而，这个假设并没有反映现代云基础设施中表现出的复杂故障原因。一个典型的云软件程序由数十个模块、数百个动态线程和数万个函数组成，用于处理不同的请求、运行各种后台任务、应用层优化等。毫不奇怪，实际中这样的程序会经历部分故障，即 其中一些而不是全部功能被破坏。例如，对于现代分布式文件系统中的数据节点进程，当该进程中的重新平衡器线程不能再

将非平衡块分配给其他远程数据节点进程时，即使该进程仍然存在，也可能会发生部分故障。或者，此数据节点进程中的块接收器守护进程静默退出，因而块不再持续地保存到磁盘。这些部分故障是运营商不容忽略的潜在问题；它们可能会造成严重损害，包括不一致、“僵尸”行为和数据丢失。事实上，部分故障是许多灾难性现实中断发生的原因 [1, 17, 39, 51, 52, 55, 66, 85, 86]。例如，Microsoft Office 365 邮件服务遭受了 8 小时的中断，而这是因为邮件服务器的反病毒引擎模块在识别一些可疑邮件时卡住了 [39]。

当发生部分故障时，通常需要很长时间才能检测到事故。相比之下，可以通过现有机制 快速识别、重新启动或修复 遭受完全故障的过程，从而降低故障影响。但糟糕的是，部分故障会导致难以调试的神秘症状 [78]，例如，create() 请求超时但 write() 请求仍然有效。在由于引导部分失败而导致产生 ZooKeeper 中断 [86] 时，即使在触发警报之后，引导日志也几乎没有关于 问题出在哪儿 的线索。开发人员花费了大量时间来定位问题引导进程中的故障（图 1）。在查明故障之前，简单重启引导进程并没有效果（因为相同症状很快又会出现）。

```
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 }
```

blocked for a long time

图 1：由于部分故障导致 ZooKeeper 中断 [86]

从业者和研究界都呼吁关注这一缺漏。例如，Cassandra 开发人员采用了更先进的应计故障检测器 [73]，但仍然得出结论，其当前设计“几乎没有能力有效地采取重要措施来处理部分故障”[13]。 Prabhakaran 等人分析了特定磁盘的部分故障 [88]。Huang 等人讨论了云基础设施中的灰色故障 [76] 挑战。然而，软件部分故障的总体特征还没有被很好地理解。

在本文中，我们首先试图回答一个问题，即 部分故障在现代系统中是如何表现的？为了阐明这一点，我们对来自五个大型开源系统的 100 个现实部分故障案例进行了研究（第 2 节）。我们发现，有近一半 (48%) 研究的故障会导致某些特定的软件功能被卡住。此外，所研究的大多数故障（71%）是由生产环境中的特有条件触发的，例如错误的输入、调度、资源争用、易碎

磁盘或有故障的远程进程。由于这些故障会影响压缩和持久性等内部特性，因此外部检测器或探测器可能无法观察到它们。

如何在运行时系统地检测和定位部分故障？从业者目前依赖于运行临时健康检查（例如，每隔几秒发送一次 HTTP 请求并检查其响应状态 [3, 42]）。但是这样的健康检查太浅了，无法暴露大范围的故障。该领域最先进的研究工作是 Panorama [75]，它将目标流程的各种请求者转换为观察者，以报告该流程的灰色故障。这种方法受到请求者从外部观察到内容的限制。此外，这些观察者无法定位故障进程中检测到的故障。

我们提出了一种新方法，通过减少程序来构建有效部分故障检测器。给定一个程序 P，我们的基本思想是从 P 派生一个简化但具有代表性的版本 W 作为检测器模块，并在生产中定期测试 W 以暴露 P 中的各种潜在故障。我们称 W 为内在看门狗。这种方法有两个主要好处：首先，由于看门狗源自主程序并“模仿”主程序，相比现有的无状态运行、浅层健康检查或外部观察者，它可以更准确地反映主程序的状态。其次，简化版本使得看门狗简明并有助于定位故障。

对于开发人员来说，在大型软件上手动应用缩减方法既耗时又容易出错。为了减轻这个负担，我们设计了一个工具，OmegaGen，它可以静态分析给定程序的源代码，并为目标程序生成定制的内在看门狗。

我们在 OmegaGen 中实现程序缩减的出发点 源自于 W 的目标仅仅是检测和定位运行时错误；因此，它不需要重新创建 P 业务逻辑的全部细节。例如，如果 P 在紧密循环中调用 write()，出于检查目的，带有一个 write() 的 W 可能足以暴露错误。此外，虽然检查各种故障很诱人，但鉴于资源有限，W 应该专注于检查仅在生产环境中出现的故障。确定性地导致错误结果的逻辑错误（例如，不正确的排序）应是离线单元测试的重点。以图 1 为例。在检查 SyncRequestProcessor 时，W 不需要检查函数 serializeNode 中的大部分指令，例如，第 3-6 行和第 8 行。虽然这些指令在生产中失败的可能性很小，但反复检查它们会为有限的资源预算产生递减收益。

一般来说，准确区分逻辑确定性故障和生产相关故障是很困难的。 OmegaGen 使用启发式方法来分析指令的“脆弱性”程度，基于指令是否执行某些 I/O、资源分配、异步等待等。因此，由于图 1 的第 9 行执行写入，因此在 W 中将被评估为易受攻击并经过测试。期望 W 总是包含故障根本原因指令是不现实的。幸运的是，一个大致的评估通常就足够了。例如，即使我们只评

估整个 serializeNode 函数或其调用者存在的漏洞，并在 W 中定期对其进行测试，W 仍然可以检测到这个部分故障。

一旦选择了易受攻击的指令，OmegaGen 会将它们封装到检查器中。 OmegaGen 的第二个贡献是提供了几个强大的隔离机制，因此看门狗检查器不会干扰主程序。对于内存隔离，OmegaGen 识别检查器的上下文，并在主程序中生成带有钩子的上下文管理器，在检查器中使用它们之前复制上下文。 OmegaGen 通过重定向消除 I/O 操作的副作用，并设计了一个幂等包装机制来安全地测试 非幂等操作。

我们已将 OmegaGen 应用于六个大型（28K 至 728K SLOC）系统。 OmegaGen 自动为这些系统生成数十到数百个看门狗检查器。为了评估生成看门狗的有效性，我们重现了 22 个现实部分故障。我们的看门狗可以检测到 20 个案例，中位检测时间为 4.2 秒，并定位了 18 个案例的故障范围。相比之下，最好的手动编写的基线检测器只能检测 11 个案例并定位 8 个案例。通过测试，我们的看门狗在最新的 ZooKeeper 中揭秘了一个新的、惯常的部分故障错误。

2. 理解局部故障

部分故障是一个普遍的问题。 Gupta 和 Shute 报告称，在 Google Ads 基础架构中，部分故障比完全故障更常见 [70]。研究人员研究了部分磁盘故障 [88] 和慢速硬件故障 [68]，但对软件是如何部分故障的却没有很好地理解 。在本节中，我们研究真实的部分故障，以深入理解这个问题并引导我们的解决方案设计。

范围 我们更多关注过程粒度上的部分故障。这个过程可以是独立的，也可以是大型服务中的一个组件（例如，存储服务中的数据节点）。我们研究的是关于 偏离部分故障本身应提供功能的过程，例如，存储和平衡数据块，不管它是服务组件还是独立服务器。我们注意到，用户可能会在服务粒度上定义部分故障（例如，Google 驱动器变更为只读），其根本原因可能是某些组件崩溃或部分故障。

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

表 1：所研究的软件系统部分故障案例 以及这些案例涵盖的特有版本和日期范围。

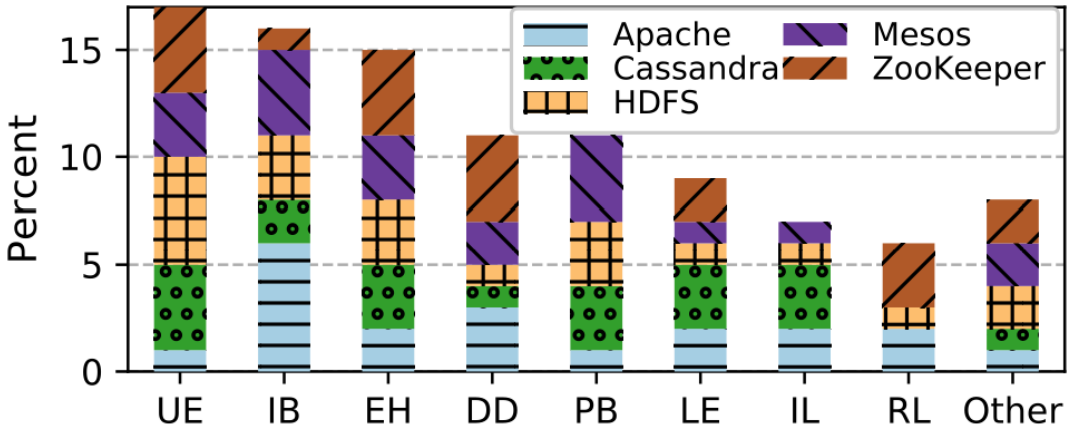


图 2：故障的根源分布。 UE：未捕获的错误； IB：无限期阻塞； EH：错误的故障处理； DD：死锁； PB：性能错误； LE：逻辑错误； IL：无限循环； RL：资源泄漏。

方法 我们研究了五个广泛使用的大型软件系统（表 1）。它们提供不同服务 并用不同语言编写。为了收集研究案例，我们首先在官方错误跟踪器中抓取所有标记为关键优先级的错误单。然后，我们从测试中过滤错误单并随机抽取剩余的故障单。为了最大限度地减少我们研究的部分故障类型的偏差，我们详尽地检查了每个抽样案例并手动确定它是否是完全故障（例如，崩溃），如果是，则丢弃。我们总共收集了 100 个故障案例（每个系统 20 个案例）。

2.1 发现

发现 1：在五个系统中，部分故障一直出现在整个版本发布史中（表 1），其中 54%发生在最近三年的软件版本中。

出现这种趋势，部分原因是 随着软件演化，新功能和性能优化被，而这使故障原因复杂化。例如，HDFS 在 0.23 版本中引入了短路本地读取功能 [30]。为了实现此功能，添加了

DomainSocketWatcher 模块，它监视一组 Unix 域套接字 并在它们变得可读时调用回调。但这个新模块可能会意外退出进程，并导致执行短路读取的应用程序挂起 [29]。

发现 2：我们研究的故障根源是多种多样的。前三种（总共 48%）根源类型是：未捕获的错误、无限期阻塞和错误的故障处理（图 2）。

未捕获的错误 是指某些特定操作触发了一些软件没有预料到的错误条件。例如，当流读取器遇到诸如 RuntimeException [6] 之类 IOException 以外的错误时，Cassandra 中的流处理过程可能会挂起。当某些函数调用被永远阻塞时，就会发生无限阻塞。在一种情况下 [27]，备用 HDFS 名称节点中的 EditLogTailer 向活动名称节点发出 RPC rollEdits()；但是当活动的 namenode 被冻结但没有崩溃时，这个调用被阻止了，这阻止了备用节点的激活。 不正确的错误处理包括静默忽略错误、空处理程序 [93]、过早继续等。其他常见的故障根源包括死锁、性能错误、无限循环和逻辑错误。

发现 3：近一半 (48%) 的部分故障导致某些功能“卡住”。

图 3 显示了所研究故障产生的后果。请注意，这些故障都是局部的。对于“卡住（或阻塞）”的故障，某些软件模块（如套接字观察程序）没有取得任何进展；但该进程并非完全无响应，即其核心模块仍能及时响应，它还可以处理其他请求，例如非本地读取。

除了“卡住”的情况外，17% 的部分故障会导致某些操作需要很长时间才能完成（图 3 中的“慢”类型）。这些缓慢故障不仅导致了可选优化措施的低效率，更进一步，作为严重的性能错误，它们导致受影响的功能几乎无法使用。在一个案例 [5] 中，将 Cassandra 2.0.15 升级到 2.1.9 后，用户发现生产集群的读取延迟从 6 ms/op 增加到 100 ms/op 以上。

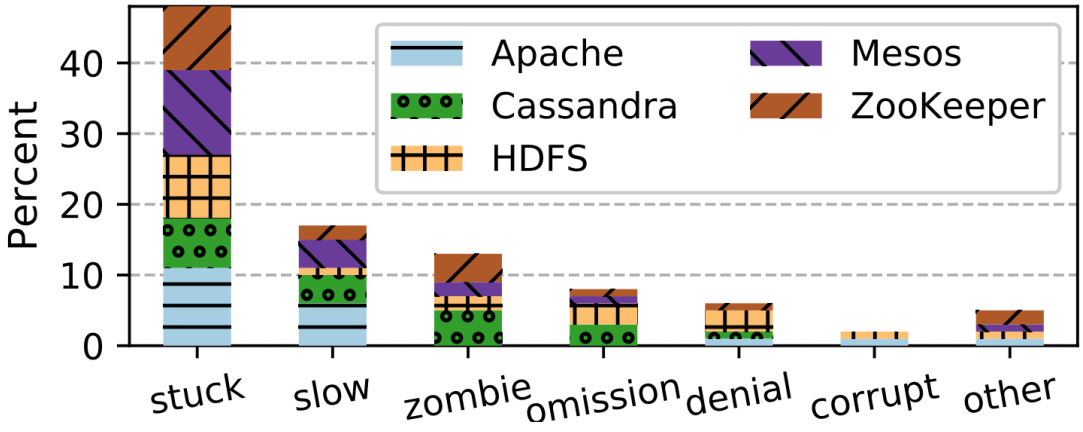


图 3：所研究故障产生的后果。

发现 4：在 13% 的研究案例中，模块变成了具有未定义故障语义的“僵尸”。

这通常发生在故障模块意外退出其正常控制循环 或 即使遇到一些无法容忍的严重错误时仍继续执行。例如，意外的异常导致 ZooKeeper 侦听器模块意外退出其 while 循环，因此新节点无法再加入集群 [46]。在另一种情况下，即使块池未能初始化 [26]，HDFS 数据节点也会继续运行，这将在尝试进行块报告时触发 NullPointerException。

发现 5：15% 的部分故障是无声的（包括数据丢失、损坏、不一致和错误结果）。

如果没有详细的正确范式，它们通常很难被检测到。例如，当 Mesos 代理收集旧从属沙箱（垃圾回收）时，它可能会错误地清除持久卷上的数据 [37]。在另一种情况 [38] 中，Apache Web 服务器会“失控”，例如，对 .js 文件的请求会收到 image/png 的响应，因为在出现错误时，不能正确关闭后端连接。

发现 6：71% 的故障是由某些特定的环境条件、输入或其他进程中的故障触发的。

例如，ZooKeeper 中的部分故障，只能在记录的长度字段中出现某些损坏时触发 [66]。另一个 ZooKeeper Leader 的部分故障只会在连接的 Follower 挂起时发生 [50]，这会阻止其他 Follower 加入集群。这些部分故障很难通过 产品产前测试 暴露出来，并且需要在运行时检测的机制。此外，如果运行时检测器使用不同的设置或检查输入，它可能无法检测到此类故障。

发现 7：大多数 (68%) 的故障是“粘性的”。

粘性意味着进程不会自行从故障中恢复。故障进程需要重新启动或修复才能再次运行。在一些情况下，竞争条件导致了意外的 RejectedExecutionException，这导致 RPC 服务器线程静默退出其循环，并停止侦听连接 [9]，必须重新启动此线程才能解决此问题。对于某些故障，需要一些额外的修复操作，例如修复文件系统不一致 [25]。

其余（32%）的故障是“暂时的”，即故障模块可能在某些条件变化后恢复，例如，当冻结的名称节点变得可响应时 [27]。然而，到那时，这些非粘性故障已经造成了很长一段时间的损坏（一种情况下为 15 分钟 [45]）。

发现 8：中位诊断时间为 6 天零 5 小时。

例如，诊断 Cassandra 故障 [10] 花了开发人员将近两天的时间。结果发现根本原因相对简单：MeteredFlusher 模块被阻塞了几分钟并影响了其他任务。尽管根本原因很简单，但诊断时间长的一个常见原因是故障 令人困惑的症状，因而误导了诊断方向。另一个常见的原因是故

障进程中运行时信息暴露不足。用户必须启用调试日志、分析堆、和/或 来检测代码，以识别产生故障期间发生的情况。

2.2 影响

总体而言，我们的研究表明，部分故障是大型软件系统中常见且影响严重的问题。大多数我们所研究的故障都与生产相关（发现 6 个），这需要运行时机制来检测。此外，如果运行时检测器除了检测之外还可以定位故障，它将降低离线诊断的难度（发现 8）。现有的检测器，如 Heartbeat、探测器 [69] 或 Observer [75] 是无效的，因为它们几乎不会暴露在流程内部受影响的功能中（例如，压缩）。

人们可能会得出结论，开发人员有责任在他们的代码中添加有效的运行时检查，例如在上述 HDFS 故障 [27] 中对 rollEdits() 操作进行计时器检查。然而，仅仅依靠开发人员预测并为每个操作添加防御性检查是不现实的。我们需要一种系统的方法来帮助开发人员构建特定于软件的运行时检查器。

完全自动化定制 运行时检查器的构建 是可取的，但考虑到部分故障的多样性（发现 2），这在一般情况下是极其困难的。事实上，我们所研究的故障中 15% 是静默的，这需要详细的正确范式才能捕获。幸运的是，大多数我们研究的故障都违反了活跃度（发现 3）或在某些程序点触发了显式错误，这表明检测器可以在没有深入语义理解的情况下自动构建。

3．使用看门狗捕获 部分故障

考虑一个由许多较小模块组成的大型服务器进程 π，它能提供一系列功能 R（如，一个具有请求侦听器、快照管理器、缓存管理器等的数据节点服务器），为了监控高可用性的过程，我们需要一个故障检测器。针对部分故障，我们专门将过程 π 中的部分故障定义为 不会导致 π 崩溃但会使某些功能 Rf（R 的安全性或活跃度）违规或严重缓慢的故障。除了检测故障外，我们的目标是定位过程中的故障，便于后续地排除和解决故障。

在研究指导下，我们提出了一个有效的交叉原则来设计部分故障检测器，即 构建 与监控过程执行相交叉的 定制检查。基本原理是部分故障通常涉及特定的软件功能和不良状态；为了暴

露此类故障，检测器需要使用精心挑选的有效载荷来运行特定代码区域。现有检测器（包括 Heartbeat 和 HTTP 测试）中的检查过于通用，并且与被监控进程的状态和执行脱节。

我们提倡在遵循上述原则下设计内在看门狗（图 4）。内在看门狗是进程的专用监控扩展，此扩展定期执行一组针对不同模块量身定制的检查器。看门狗驱动程序管理检查程序的调度和执行，并选择应用恢复操作。检测的关键目标是让看门狗遇到与主程序类似的故障，而这可以通过：(a) 执行模仿式检查器 (b) 使用有状态的有效负载 (c) 共享受监视进程的执行环境，来实现。

模仿式检查器。现在的检测器 使用两种类型的检查器：（1）探测检查器，它定期调用一些 API；（2）信号检查器，它监控一些健康指标。两者都是轻量程的。但是探针检查器可能会漏掉许多故障，因为大型程序有许多 API，并且在 API 级别可能无法观察到部分故障。而信号检查器容易受到环境噪声的影响，并且通常精度较差。此外，两者都不能定位所检测到的故障。

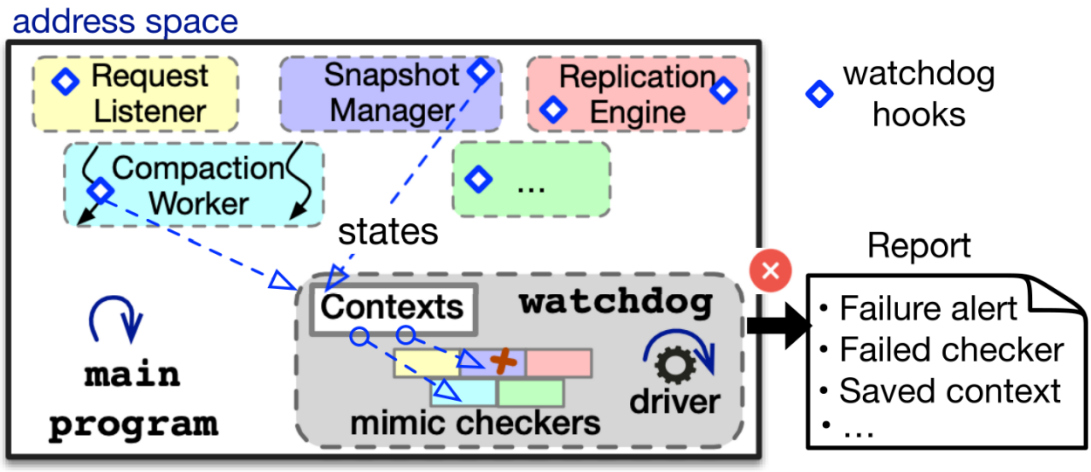


图 4：内在看门狗示例。

我们提出了一个更强大的模仿式检查器/模拟检查器。这种检查器从主程序的每个模块中选择一些有代表性的操作，对其进行模仿/模拟，并检测错误。这种方法增加了检查目标的覆盖范围。并且由于检查器在生产环境中采用类似于主程序的代码逻辑，因此可以准确地反映被监控进程的状态。此外，模拟检查器可以查明有故障的模块和失败的指令。

同步状态。锻炼检查器需要有效载荷。现有的检测器使用合成输入（例如，固定 URL [3]）或程序状态的一小部分（例如，Heartbeat 变量）作为有效负载。但是触发部分故障通常需要特定的输入和程序状态（§2），看门狗应该在主程序中使用非普通状态的检查器来提高暴露部分故障的机会。

我们在看门狗中引入上下文。上下文绑定到每个检查器，并保存检查器执行所需的所有参数。上下文通过主程序中的钩子与程序状态同步。当主程序执行到达挂钩点时，挂钩使用当前程序状态更新其上下文。除非它的上下文准备好，否则看门狗驱动程序不会执行检查器。

并发执行。在主程序中直接插入检查器是很自然的。然而，内部检查带来了固有压力——一方面，捕捉部分故障需要添加全面的检查器；另一方面，部分故障很少发生，但是在正常情况下，更多的检查器会减慢主程序的速度。内部检查器还可以通过修改程序状态或执行流程轻松干扰主程序。

我们提倡看门狗与主程序同时运行。由于并发执行 允许解耦式检查，因此看门狗可以执行全面的检查，而不会在程序正常执行期间延迟主程序。事实上，嵌入式系统领域早已探索使用并发看门狗协处理器技术来进行有效的错误检测[84]。当检查器触发一些错误时，看门狗也不会意外地改变主程序的执行。此外，并发看门狗应位于相同的地址空间中，以最大限度地模拟执行并暴露类似的问题，例如，当进程遇到长时间的 GC 暂停时，所有检查器都会超时。

4．使用 OmegaGen 生成看门狗

为大型程序手动编写有效的看门狗很单调，而且很难做到准确无误；而不严谨编写的看门狗可能会遗漏检查重要功能、更改主执行、调用危险操作、破坏程序状态等；此外，看门狗也必须随着软件的发展而更新。为了减轻开发人员的负担，我们设计了一个工具 OmegaGen，它使用新颖的程序缩减方法来自动生成第 3 节中描述的看门狗。OmegaGen 的核心挑战是确保生成的看门狗准确反映主程序状态，而不会引入显著的开销或副作用。

概述和目标。 OmegaGen 将程序 P 的源代码作为输入。它在 P 中找到长时间运行的代码区域，然后使用启发式方法、可选用户提供的注释 来识别可能遇到的与生产相关问题的指令。OmegaGen 将易受攻击的指令封装到可执行检查器中，并生成看门狗 W；它还在 P 中插入看门狗钩子，以更新 W 的上下文并打包驱动程序，从而在 P 中执行 W。图 5 显示了运行 OmegaGen 的概述示例。

如第 2.2 节所述，自动生成可以捕获所有类型的部分故障的检查器是困难的。我们的检查器生成方法，针对这样一类部分故障，它们是由于程序中某些指令或功能的显式错误、阻塞或缓慢而导致的。 OmegaGen 生成的看门狗在捕捉部分故障方面特别有效，例如程序运行时某些模块卡住、非常慢或“僵尸”（例如，HDFS DomainSocketWatcher 线程意外退出并影响短路读取）。

此外，OmegaGen 生成的看门狗通常对静默正确性错误无效（例如，Apache Web 服务器错误地重用过时的连接）。

4.1 识别长时间运行的方法

OmegaGen 通过识别程序中长时间运行的代码区域（步骤 1）开始其静态分析，因为看门狗只针对连续执行的代码进行检查。服务器程序中的许多代码区域仅用于一次性任务，例如数据库创建，则应该从看门狗中排除。有些任务要么定期执行，例如快照，要么仅在特定条件下激活。我们需要确保看门狗的启动，与其在主程序中检查目标的生命周期一致。否则，它可能会报告错误的检测结果。

OmegaGen 遍历程序将调用图中的每个节点。对于每个节点，它识别函数体中可能长时间运行的循环，例如，while(true) 或 while(flag)；而具有固定迭代或迭代集合的循环将被跳过。然后 OmegaGen 在已识别的循环体中定位所有的调用指令。被调用的目标将被着色，任何由彩色节点调用的方法也都是递归着色的。除了循环之外，我们还支持着色周期性任务方法，计划通过 Java 并发包中的 ExecutorService 等常用库。请注意，此步骤可能会过度提取（例如，条件下的调用）。但这不是大问题，因为看门狗驱动程序将在运行时检查上下文有效性（第 4.4 节）。

(a) A module in main program **(b)** Generated checker

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

(a) 主程序中的一个模块 (b) 生成的检查器

图 5：为 ZooKeeper 中的模块生成的看门狗检查器 OmegaGen 示例。

当一个方法有多个调用点时会出现复杂情况，其中一些被着色，而另一些则没有。此方法是否长时间运行取决于具体执行情况。此外，一个已识别的长时间运行循环，在实际运行中可能会变成短暂运行的。为了准确捕获方法生命周期并控制看门狗激活，OmegaGen 设计了一种基于谓词的算法。谓词是与方法关联的运行时属性，用于跟踪是否确实到达了该方法的调用点。

对于可能长时间运行的循环内的调用目标，OmegaGen 在设置其谓词的循环之前插入一个钩子，并在取消设置其谓词的循环之后插入另一个钩子。一个潜在长时间运行方法的被调用者，将有一个等效于该调用者谓词的谓词集合。在运行时，OmegaGen 将分配和评估谓词集合，而这会激活或停用相关看门狗。在 OmegaGen 完成易受攻击的操作分析（第 4.2 节）和程序缩减（第 4.3 节）之后，谓词检测发生。

4.2 定位易受攻击的操作

然后 OmegaGen 会分析识别长时间运行的方法，并进一步缩小检查目标的候选范围（步骤 2）。这是因为即使这些方法数量有限，看门狗也无法检查所有的操作。我们的研究表明，大多数部分故障是由独特的环境条件或工作负载触发的。这意味着那些安全性或活跃性受其执行环境严重影响的操作，值得特别关注。相比之下，逻辑上正确性确定的操作（例如排序），最好通过离线测试或本地确认来检查；在看门狗内部持续监控此类操作反而将产生递减收益。

OmegaGen 使用启发式方法来确定给定操作在其执行环境中的脆弱程度。目前，启发式方法认为执行同步、资源分配、事件轮询、异步等待、使用外部输入参数调用、文件或网络 I/O 的操作非常容易受到攻击，而 OmegaGen 通过标准库调用识别其中大部分易受攻击的操作。由于潜在的无限循环，包含复杂 while 循环条件的函数被认为是易受攻击的；而诸如算术、赋值和数据结构字段访问等简单操作被标记为不易受攻击。在图 5a 示例中，Omega Gen 认为 oa.writeRecord 非常容易受到攻击，因为它的主体调用了多个写入请求。我们的研究将提供这些启发式方法，但也可以通过 OmegaGen 中的规则表配置进行定制。例如，我们可以将 OmegaGen 配置为 将具有多个异常特征的函数视为易受攻击的（即，潜在不当处理）；我们还允许开发人员在源代码中使用 @vulnerable 标记注释方法。 OmegaGen 将定位对带注释方法的调用，并将它们视为易受攻击的。

无论是我们的启发式方法还是人工判断，都不能始终保证 易受攻击操作的标准 是健全和完整的。如果 OmegaGen 错误地将安全操作评估为易受攻击，带来的主要后果是 看门狗会浪费资

源来监控不必要的东西；而若将易受攻击的操作错误地评估为无风险 则更令人担忧。但是易受攻击操作有一个很好的特征,即 它们通常是传播性的[67]——如 无限期阻塞的指令也会导致其封闭函数阻塞；并且，触发某些未捕获错误的指令也会通过调用堆栈传播。例如，实际在 ZooKeeper [66] 的部分故障中，即使 OmegaGen 错过了确切的易受攻击指令 readString，但如果 dserialize 甚至 pRequest 被评估为易受攻击，看门狗仍然有机会检测到部分故障。另一方面，如果一个易受攻击的操作级别太高（例如，main 被认为是易受攻击的），错误信号可能会在内部被吞下，并且也会使定位故障变得困难。

4.3 缩减主程序

随着识别出 长时间运行方法和易受攻击的操作,OmegaGen 从长时间运行方法的入口点开始执行自上而下的程序缩减(步骤 3)。例如,在图 5a 中,OmegaGen 将首先尝试减少 takeSnapshot 函数。当遍历待约简方法的控制流图时，如果一条指令被标记为潜在易受攻击的，它将保留在约简方法中，否则，它将被排除在外；对于尚未标记为易受攻击的调用指令，它将被暂时保留，并且 OmegaGen 将递归地尝试减少目标函数。如果最终约简方法的主体为空，即不存在易受攻击的操作，它将被丢弃。任何调用此丢弃方法（指最终约简为空的方法）并暂时保留的调用指令也将被丢弃。

生成的简化程序不仅包含所有易受攻击操作（它们是从长时运行方法中可达的），而且还保留了原始结构，即对于主程序中的调用链 f → g → h ，简化后的调用链为 f’ → g’ → h’ 。这个结构可以帮助本地化已记录的问题。此外，稍后当看门狗调用验证器（第 4.6 节）时，该结构将提供有关调用哪个验证器的信息。

如果一种易受攻击的操作（例如，图 5a 中的 writeRecord 调用）多次包含在约简程序中，则在暴露故障方面可能是多余的。因此，OmegaGen 将根据是否已被收录，进一步减少易受攻击的操作。但是，同一类型的易受攻击操作在不同的地方可能会被完全不同地调用，并且只有特定的调用会触发故障。如果我们过于激进地根据出现率削减（重复出现的易受攻击操作），我们可能会错过触发故障的调用。因此，默认情况下，OmegaGen 只执行过程内发生的削减：如 多个 writeRecord 调用不会发生在单个需要削减的方法中，但可能发生在不同的需削减方法中。

4.4 封装精简程序

OmegaGen 会将步骤 3 之后保留的代码片段封装到看门狗中。但由于缺少定义或有效负载，这些代码片段可能无法直接执行。例如，图 5a 中的简化版 serializeNode 中包含一个操作 oa.writeRecord(node, "node")，但 oa 和 node 是未定义的。 OmegaGen 将分析所有执行约简方法所需的参数。对于每个未定义的变量，OmegaGen 在约简方法的开头添加一个局部变量定义。它进一步生成一个上下文代理，该代理提供 API 来管理约简方法中的所有参数（步骤 4）。在约简方法中第一次使用变量之前，会添加对上下文代理的 getter 调用，以在运行时检索最新值。

为了与主程序同步，OmegaGen 在原始程序（非约简）方法中，插入同一访问点、同一上下文代理中 调用 setter 方法的钩子。上下文挂钩则更进一步取决于此方法的长时运行谓词（第 4.1 节）。当看门狗驱动程序执行约简方法时，它首先检查上下文是否准备好，如果上下文未准备好则跳过执行。上下文和谓词一起控制看门狗检查器的激活——只有当原始程序到达上下文挂钩并且该方法真正长时运行时，才会检查相应的操作。例如，在图 5a 的 while 循环中，如果日志计数尚未达到快照阈值，虽然 takeSnapshot 的谓词为真，但约简化 serializeNode 的上下文尚未准备好，因此跳过检查。

4.5 添加检查以捕获故障

在步骤 4 之后，封装的约简方法可以在看门狗中执行。然后，OmegaGen 将添加对看门狗驱动程序的检查，在约简方法中执行易受攻击的操作，以从中捕获故障信号。 OmegaGen 同时针对活跃度违规和安全性违规行为。添加活跃度检查相对简单，OmegaGen 在运行检查器之前会插入一个计时器。众所周知，为分布式系统设置良好的超时设定是一个难题。先前的工作 [82] 认为，用本地操作详细深入的超时来替换端到端超时会使设置不那么敏感。我们进行了类似的观察并使用了保守的超时时间（默认为 4 秒）。除了超时，看门狗驱动程序还记录检查器执行延迟的移动平均值，以检测潜在的慢故障。

为了检测安全性违规，OmegaGen 依靠易受攻击的操作来发出明确的错误信号（断言、异常和错误代码），并安装处理程序来捕获它们。 OmegaGen 还捕获运行时错误，例如空指针异常、内存不足错误、非法状态异常。

在不了解易受攻击操作语义的情况下，更难自动检查正确性违规。幸运的是，在我们研究的案例中，这种静默的违规行为并不常见（§2）。尽管如此，OmegaGen 还是为开发者提供了一个 wd_assert API 来方便地添加语义检查。 OmegaGen 分析程序时，会将 wd_assert 指令视为特

殊的易受攻击操作。它通过分析此类操作所需的上下文，执行类似的检查器封装（第 4.4 节），并生成包含 wd_assert 指令的检查器。主程序中原来的 wd_assert 将被重写为空操作。通过这种方式，开发人员可以利用 OmegaGen 框架执行并发的、代价昂贵的检查（例如，新块的哈希表与检查器的 校验和 匹配），而不会阻塞主程序执行。

看门狗驱动程序将在日志文件中记录任何检测到的错误。报告的错误包含时间戳、故障类型和症状、失败的检查器、失败的检查器正在测试的相应主程序位置、回溯等。看门狗驱动程序还保存了失败检查器使用的上下文，以方便后续的脱机故障排除。

4.6 验证捕获故障的影响

看门狗检查器报告的错误可能是暂时性或可容忍性的。为了减少误报，看门狗在检测到错误后运行验证任务。默认验证是简单地重新执行检查器和比较，这对暂时性错误有效；而验证可容忍性错误则需要测试软件功能。请注意，验证器不是用于处理错误，而是用于确认影响。编写此类验证任务主要涉及调用一些入口函数，例如 processRequest(req)，这很简单。

OmegaGen 提供验证任务的框架，目前依靠人工来填写框架。但是 OmegaGen 会根据哪个检查器失败来自动选择调用哪个验证任务。具体来说，对于在主程序中调用函数 f 的填充验证任务 T，OmegaGen 按拓扑顺序搜索生成的约简程序结构（第 4.3 节），并尝试找到第一个与 f 或 f 调用图中的任何方法匹配的约简方法 m'。然后 OmegaGen 会生成一个哈希图，将所有以 m' 为根的检查器映射到任务 T 。在运行以报告错误时，看门狗驱动程序会检查映射以决定调用哪个验证器。

4.7 防止副作用

上下文复制。为了防止看门狗检查器意外修改主程序的状态，OmegaGen 会分析检查器中引用的所有变量（上下文）。在检查器的上下文管理器中 OmegaGen 将生成一个拷贝设置器——它会在调用时复制上下文，复制确保了任何修改都会包含在看门狗的状态中；使用复制上下文还可以避免在检查期间添加复杂的同步来锁定对象，但是盲目地复制上下文也会产生很高的开销。我们对看门狗上下文进行不变性分析 [74, 77]，如果上下文是不可变的，OmegaGen 会生成一个引用设置器，它只保存对上下文源的引用。

为了进一步减少上下文复制，我们使用了一种简单而有效的惰性复制方法，它并非在每个集合上复制一个上下文，而是将复制延迟到仅当 getter 函数需要它时。为了处理由于延迟复制导致的潜在不一致性（例如，主程序在 setter 调用后修改了上下文）——我们将上下文与几个属性相关联：版本、weak_ref（对源对象的弱引用）和 hash（用于源对象的值）。惰性设置器仅设置这些属性，但不复制上下文。稍后当调用 getter 时，getter 会检查 weak_ref 的所指对象是否不为空；如果是，它会进一步检查参考值的当前哈希码是否与记录的哈希匹配；如果不匹配则跳过复制（主程序修改上下文）。除了 getter 中的属性检查外，看门狗驱动程序还将检查易受攻击操作中每个上下文的版本属性是否匹配，如果版本不一致则跳过检查（参见附录 A 中的详细说明）。

I/O 重定向和幂等包装器。除了存储副作用；我们还需要防止 I/O 副作用。例如，如果一个易受攻击的操作正在写入快照文件，则看门狗可能会意外写入同一快照文件并影响主程序的后续执行。 OmegaGen 在看门狗中添加了 I/O 重定向功能来解决这个问题：当 OmegaGen 生成上下文复制代码时，复制过程将检查上下文是否引用了相关文件的资源，如果是，则上下文将与文件路径一起复制改成同目录路径下的看门狗测试文件。因此，看门狗会遇到类似的问题，例如退化或存储故障。

但是，如果正在写入的存储系统是内部负载平衡的（例如 S3），则测试文件可能会分发到不同的环境中，从而错过仅影响原始文件的问题。由于我们的写入重定向是在克隆库中实现的，因而可以解决这个局限，也因此扩展决定重定向路径的逻辑以考虑负载平衡策略（若风故障风险高）相对容易。此外，如果底层存储系统像 S3 那样分层且复杂，那么在该系统上应用 OmegaGen 以直接暴露那里的部分故障可能会更好。

对于套接字 I/O，如果我们事先知道远程组件也是 OmegaGen 检测的，那么 OmegaGen 可以执行类似的重定向到特殊的看门狗端口。由于这个假设可能不成立，OmegaGen 默认将看门狗的套接字 I/O 操作重写为 ping 操作。

如果易受攻击的操作是读取类型操作，重定向到从看门狗特殊测试文件中读取可能无济于事。因此我们设计了一种幂等包装器机制，以便主程序和看门狗都可以安全地调用包装器。如果主程序首先调用包装器，它会直接执行实际的读取类型操作并将结果缓存在上下文中。而当看门狗调用包装器时，如果主程序在临界区，它会等到主程序完成，然后获取缓存的上下文。在正常情况下，看门狗可以使用读取操作中的数据，而无需执行实际读取。在故障场景中，如果主程序

无限期地阻塞在执行读类操作，看门狗会通过在其包装器中由等待超时来发现挂起问题；读取的错误值也将在检索后被看门狗捕获。对于每一个易受攻击的 read-type 操作，OmegaGen 都会生成一个具有上述属性的幂等包装器，将主程序的 原始调用指令 替换为 调用包装器 ，并在看门狗检查器中放置一个包装器的调用指令。

5. 实现

我们用 8,100 源代码行的 Java 实现了 OmegaGen。它的核心组件建立在 Soot [90] 程序分析框架之上，因此它支持 Java 字节码系统。 OmegaGen 不依赖于特定的 JDK 功能，其使用的 Soot 版本可以分析高达 Java 8 的字节码。我们利用一个克隆库 [79] 更改了大约 400 源代码行，以支持我们进行 选择性上下文复制 和 I/O 重定向机制。 OmegaGen 的工作流程由多个阶段组成，用于分析和检测程序并生成看门狗。单个脚本使工作流程自动化，并将看门狗与主程序打包成一个包。

	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.

表 2：评估的系统软件。 ZK：一种开源的分布式应用程序协调服务； CS：一个来自 Apache 的分布式数据库，具有高度可扩展性，可用于管理大量的结构化数据；HF：Hadoop 分布式文件系统； HB：一个开源的分布式 NoSQL 数据库；MR：面向大数据并行处理的计算模型、框架和平台； YN：一个快速、可靠和安全的 JavaScript 依赖管理工具。

6 .评估

我们评估 OmegaGen 是为了回答几个问题：(1) 我们的方法是否适用于大型软件？ (2) 生成的看门狗能否检测和定位现实中各种形式的部分故障？ (3) 看门狗是否提供强隔离？ (4) 看门狗是否报告误报？ (5) 主程序的运行时开销是多少？我们的实验是在 10 个云虚拟机的集群上进行的，每个 VM 有 4 个 2.3GHz 的 vCPU、16 GB 内存和 256 GB 磁盘。

6.1 生成看门狗

为了评估我们提出的技术是否适用于现实世界中的软件，我们在六个大型系统上评估了 OmegaGen（表 2）。我们之所以选择这些系统，是因为它们应用广泛且具有代表性，其代码库大至 728K 源代码行可供分析。 OmegaGen 使用大约 30 行默认规则用于易受攻击的操作启发式（大多数是 Java 库方法的类型）和平均 10 条系统特定规则（例如，特殊的异步等待模式）。OmegaGen 成功地为所有六个系统生成了看门狗。

表 3 显示了生成的总的看门狗。这里的每个看门狗都是约简方法的根源；要注意的是，这些都是静态看门狗。看门狗谓词和上下文挂钩（第 4.1 节）将在生成过程中激活其中的一个子集。我们通过测量软件中有多少线程类，至少生成了一个看门狗检查器，进一步评估了生成检查器的全面性。图 6 显示了结果，OmegaGen 的平均覆盖率达到 60%。而对于没有检查器的线程，它们要么不是长时间运行的（例如辅助工具），要么 OmegaGen 没有在其中发现易受攻击的操作。一般来说，OmegaGen 可能无法为主要执行计算或数据结构操作的模块生成好的检查器，即使在缩减（第 4.3 节）之后，生成的检查器仍可能包含一些冗余。

	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.

表 3：生成的看门狗和检查器的数量，并非所有看门狗都会在运行时激活。

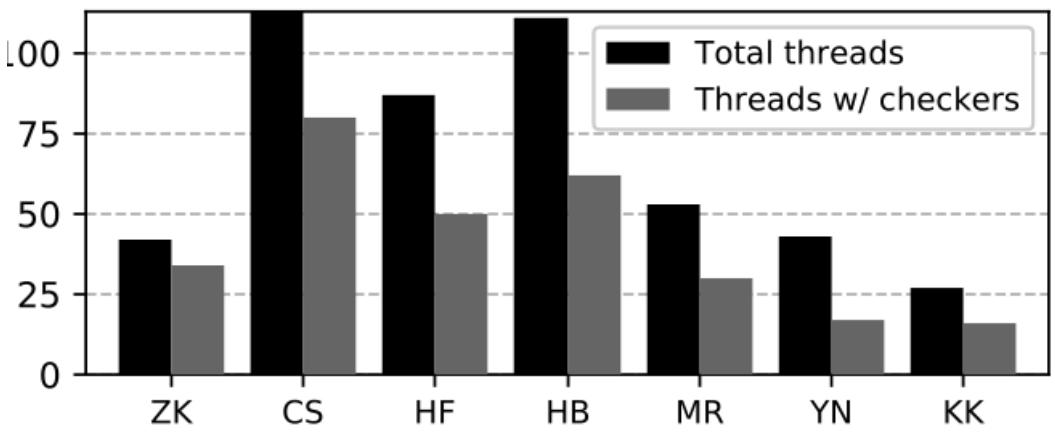


图 6：生成的看门狗检查器的线程级覆盖率。

6.2 检测现实世界的部分故障

故障基准 为了评估所生成看门狗的有效性，我们在六个系统中收集并重现了 22 个真实世界的部分故障。附录中的表 10 列出了故障案例和类型。所有这些故障都导致了严重的后果，这些故障涵盖了复杂的故障引入和工作负载触发，我们平均需要 1 周的时间来重现每个故障。其中有七个案例来自我们在第 2 节中的研究，其他则是我们以前没有研究过的新案例。

基准检测器 六个系统中的内置检测器（Heartbeat）根本无法处理部分故障。因此，我们实现了四种类型的高级检测器进行比较（表 4）:客户端检查器基于最先进的 Panorama [75] 工作中的观察者；探针检查器呈现 Falcon [82] 应用程序间谍（它们也是在 Falcon 论文中手动编写的）；在实现信号和资源检查器时，我们遵循当前的最佳实践 [15, 42] 并监控从业者推荐的信号 [2, 31, 41, 43]。

探测器	说明
客户端 (Panorama [75])	检测和监控客户端响应
探测（Falcon [82]）	进程中的守护线程，周期性地使用合成请求调用内部函数
信号	扫描日志并检查 JMX [40] 指标的脚本
资源	监视内存使用、磁盘和 I/O 健康状况以及活动线程计数的守护线程

表 4：我们所实施的四种基线检测器。

方法 看门狗和基准检测器都设置为每秒运行检查。在重现每个案例时，我们会记录软件何时到达故障程序点，以及检测器何时首次报告故障，检测时间是后者减去前者。对于缓慢故障，很难选择一个精确的开始时间;我们使用从业者推荐的标准来设置起点,例如,当 ZooKeeper [31] 的未完成请求数量超过 10 时。

结果 表 5 显示了结果。总体而言，看门狗(Watch)在 22 例故障中检测到 20 例，中位检测时间为 4.2 秒。检测到的案例中有 12 个被默认的易受攻击操作规则捕获， 8 个被系统特定的规则捕获。一般来说，看门狗对死锁、无限期阻塞以及触发显式错误信号或异常的安全问题等活跃度问题非常有效，但是它们对于 静默的正确性错误 检测效果较差。

相比之下，如表 5 所示，最好的基准检测器仅检测到 11 例，即使是所有基准检测器的组合也仅检测到 14 例。客户端检查器错过了 68% 的故障，因为这些故障涉及内部功能或一些客户端无法立即看到的优化。信号检查器（Signal）是基准检测器中最有效的，但它也很易受干扰（第 6.6 节）。

	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.

表 5：表 10 中真实故障案例的检测时间（以秒为单位）， x ：未检测到。

案例研究 ZK1 [45]：这是论文中的运行示例。网络问题导致 ZooKeeper 远程快照转储操作在关键部分被阻止，从而阻止了更新类型请求处理线程的继续进行（图 1）。 OmegaGen 生成一个检查器 serializeNode_reduced，它在 4 秒内就检测出故障。

	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.

表 6：表 10 中实际案例的故障定位。 ➳：查明故障仪器。 ✻：查明错误的函数或数据结构。 ✺：查明故障函数调用链中的函数。 ◗：查明程序中的某个入口函数，这与故障的根本原因相去甚远。 ●：仅查明错误的过程。 ❍：误导性地指出另一个无辜过程。 n/a：不适用，因为未检测到故障

CS1 [7]：Cassandra Commitlog 执行器由于提交磁盘卷错误而意外终止。这导致未提交日志的写入堆积，进而导致大量垃圾收集和进程陷入僵尸状态。 OmegaGen 生成的相关看门狗是 CommitLogSegment_reduced，有趣的是，这个案例的检测时间为负。发生这种情况，是因为执行器在故障之前成功执行了故障程序点并设置了看门狗上下文（日志段路径）；当检查器被列入进程时，上下文仍然有效，因此检查器被激活并提前暴露了问题。

HB5 [18]：即使启用了 WAL (Write Ahead Log，预写日志)滚动，用户也会在其 HBase 集群上观察到一些庞大的预写日志 (WAL)。这是因为先前当一个对等点被删除时，线程会被阻塞以向关闭的执行程序发送关闭请求。不幸的是，这个过程持有相同的锁 ReplicationSourceManager#recordLog，它将执行 WAL 滚动（为了截断日志）。我们生成的看门狗模拟了提交请求和等待完成的过程，并且在关闭的执行器上遇到了同样的停滞问题。

CS4 [11]：由于 Cassandra 压缩模块中的严重性能错误，为分区创建的所有已过期的 RangeTombstone 将保留在内存中，直到压缩完成。当工作负载包含大量对集合的覆盖时，压缩任务会非常慢。 OmegaGen 生成的相关检查器是 SSTableWriter#append_reduced。在墓碑文件堆积起来后，此检查器会根据操作延迟（平均漂移线）的急剧增加（10 倍）报告缓慢警报。

YN1 [44]：新应用程序 (AM) 在分配给最近添加的 NodeManager (NM) 后被卡住。这是由于 ResourceManager (RM) 上的 /etc/hosts 未更新造成的，因此当 RM 构建服务令牌时，这个新的 NM 无法解析。而 RM 将永远重试，而 AM 将继续分配给同一个 NM 。我们的监管机构未能检测到该问题，原因是错误操作 buildTokenService() 主要创建了一些数据结构，因此 OmegaGen 未能将其视为易受攻击。

6.3 部分故障定位

部分故障检测只是第一步，我们进一步评估了表 5 中检测到的故障案例定位的有效性。我们测量了错误报告位置和错误程序点之间的距离，并将距离分为六个降低精度的级别，表 6 显示了结果。看门狗直接为 55% (11/20) 的检测案例查明错误指令，这证明了易受攻击操作启发式方法的有效性。在案例 MR1 [35] 中，在注意到异常症状后（reducer 长时间没有进展），用户花了两天多的时间仔细分析日志和线程转储，以缩小故障原因范围。而有了看门狗错误报告后，故障就很明显了。

对于 35% (7/20) 的检测案例，看门狗定位到同一函数内的某个程序点或调用链中的某个函数，这仍然可以显著简化故障排除。例如，在 HF2 [24] 的情况下，平衡器在 waitForMoveCompletion() 中陷入循环，因为当没有可用的移动线程时， isPendingQEmpty() 将返回 错误。虽然生成的看门狗没有精确定位任何一个地方，但它在检查器 dispatchBlockMoves_reduced 中执行 future.get() 易受攻击的操作时超时从而发现了错误，因而缩小了故障范围。

相比之下，客户端或资源检测器只能查明故障进程。为了缩小故障范围，用户必须花费大量时间分析日志和代码。在 HB4 [21] 的情况下，客户端检查员甚至将故障归咎于无辜过程，这将完全误导诊断。探针检查器将故障定位到程序中的某些内部功能，但这些功能仍然太高级太抽象，离发现故障还很远。信号检查器则定位了 8 个案例。

6.4 故障引入测试

为了评估看门狗在实际部署中的表现，我们在最新的 ZooKeeper 上进行了随机故障注入（引入）实验。具体来说，我们向系统注入了四种类型的故障： 无限循环（修改循环条件以强制其永远运行）；任意延迟（在一些复杂操作中注入 30 秒延迟）；系统资源争用（耗尽 CPU/内存资源）； I/O 延迟（在文件系统或网络中注入 30 秒延迟）。之后，我们运行一系列工作负载和操作（例如，重新启动某些服务器）。我们成功触发了 16 次综合故障。我们生成的看门狗可以检测到 16 个触发的综合故障中的 13 个，中位检测时间为 6.1 秒。看门狗确定了 11 个案例的故障引入范围。

	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.

表 7：评估的六个系统中所有探测器的误报率 (%)。每个单元格报告三种设置（稳定、加载、可容忍设置）下的误报比率范围。 watch_v：带有验证器的看门狗。

6.5 发现一个新的部分故障错误

在持续测试中，我们的看门狗在 ZooKeeper 的最新版本（3.5.5）中暴露了一个新的部分错误。我们观察到 ZooKeeper 集群偶尔会挂起，以及当新的创建请求超时时，管理工具仍然显示引导进程正在工作。这种症状类似于我们研究的 错误 ZK1，但该错误已在最新版本中修复。这

个新的部分错误也是非决定性的，看门狗会在 4.7 秒内报告故障。看门狗日志帮助我们查明这个令人费解的故障的根本原因。日志显示报告问题的检查器是 serializeAcls_reduced。我们进一步检查了这个函数，发现问题在于服务器在临界区中序列化 ACLCache。当开发人员修复 ZK1 错误时，这个类似的缺陷被忽略了，而最近对 该类的重构 使该缺陷更加成问题。我们报告了这个新的错误 [49]，该错误已得到开发人员的确认并已修复。

6.6 副作用以及误报

运行具有广泛工作负载的看门狗增强系统，我们验证系统并通过了自己的测试。我们还通过将文件和客户端响应与原始系统中的文件进行比较，以验证它们的完整性。但是，如果我们禁用副作用预防机制（第 4.7 节），系统将出现明显异常，例如，快照损坏、系统崩溃；或者，主程序会挂起，因为看门狗从流中读取数据。

我们在三种设置下进一步评估看门狗和基准检测器的误报： 稳定设置：在中等工作负载下无故障运行 12 小时（§6.7）；加载设置：随机节点重启，每 3 分钟进入中等工作负载，切换到激进工作负载（3 倍客户端数量和 5 倍请求大小）；可容忍设置：以系统可容忍的引入瞬态错误运行。表 7 显示了结果，误报率是根据总的错误失败报告除以检查执行的总数来计算的。看门狗在稳定设置中没有报告错误警报。但是在加载期间，由于套接字连接错误或资源争用，它们会产生大约 1% 的误报。一旦瞬态故障消失,这些误报就会消失。使用验证器机制(第 4.6 节)，看门狗误报率（watch_v 行）显著降低。在基准检测器中，我们可以看到，虽然信号检查器实现了更好的检测，但它们也会产生高误报率（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).

表 8：OmegaGen 看门狗生成时间（秒）。

	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.

表 9：使用不同检测器的系统吞吐量 (op/s)。

6.7 性能和开销

我们首先测量 OmegaGen 静态分析的性能。表 8 显示了结果。除了 HBase 之外，整个过程不到 5 分钟。 HBase 需要 17 分钟来生成看门狗，因为它的代码库很大。

我们接下来测量启用看门狗和基准检测器的运行时开销。我们使用如下的流行基准配置：对于 ZK，我们使用开源基准 [16]，其中 15 个客户端发送 15,000 个请求（40% 读取）；对于 Cassandra，我们使用 YCSB [61] 和 40 个客户端发送 100,000 个请求（50% 读取）；对于 HDFS，我们使用了内置的基准 NNBenchWithoutMR，它创建和写入 100 个文件，每个文件有 160 个块，每个块为 1MB；对于 HBase，我们使用 YCSB 和 40 个客户端发送 50,000 个请求（50% 读取）；对于 MapReduce 和 Yarn，我们使用了内置的 DFSIO 基准测试，它写入了 400 个 10MB 文件。

表 9 显示看门狗在吞吐量上产生 5.0% - 6.6% 的开销。主要开销来自看门狗挂钩而不是并发检查器执行。探针检测器更轻量级，产生 0.2% 至 3.2% 的开销。我们还测量了延迟影响。看门狗在平均延迟上产生 9.3% - 12.2% 的开销，在尾部（第 99 个百分位）延迟上产生 8.3% - 14.0% 的开销。但鉴于看门狗在故障检测和定位方面的显著优势，我们认为其较高的开销是合理的。对于云基础设施，运营商还可以选择在已部署节点的子集上激活看门狗，以减少开销，同时仍能实现良好的覆盖范围。

我们测量带看门狗和不带看门狗的每个系统的 CPU 使用率。结果为 57%→66%（ZK）、199%→212%（CS）、33%→38%（HF）、36%→41%（HB）、5.6%→6.9%（MR）、1.5% →3% (YN)。我们还分析了堆内存使用情况。内存使用中位数（以 MiB 为单位）为 128→131 (ZK)、447→459 (CS)、165→178 (HF)、197→201 (HB)、152→166 (MR)、154→157 (YN) 。与主程序中的连续对象分配相比，增加的幅度很小，因为只在每个检查间隔延迟复制上下文。

6.8 灵敏度

我们评估 在默认 4 秒超时阈值下， 检测 ZK1 [45]（卡住故障）和 ZK4 [48]（慢故障）的活跃度问题时 的故障检测灵敏度。在超时阈值 100 ms、300 ms、500 ms、1 s、4 s 和 10 s 下，ZK1 的检测时间分别为 0.51 s、0.61 s、0.70 s、1.32 s、4.28 s 和 12.09 s。检测时间通常随着超时时间的减小而减少，但它受检查间隔的限制。超时时间为 100 毫秒时，我们在 5 分钟内观察到 6 个误报。对于 ZK4，当超时阈值激进时，可以在没有 移动平均机制（§4.5）的情况下检测到慢故障，特别是检测时间为 61.65 秒（超时阈值 100 毫秒）、91.38 秒（超时阈值 300 毫秒）、110.32 秒（超时阈值 500 毫秒）时。最终，资源泄漏会在看门狗超过更保守的阈值之前耗尽所有可用内存。

7．局限性

OmegaGen 中有几个我们计划在未来工作中解决的局限性：(1) OmegaGen 看门狗生成器的易受攻击操作的分析是基于启发式的。这一步可以通过离线分析或动态自适应选择来改进。(2) 所生成的看门狗对活跃度问题和常见的安全违规行为有效。但是它们对于捕捉静默的语义故障是无效的。我们计划利用包含语义提示（例如测试用例）的现有资源来派生运行时语义检查。 (3) OmegaGen 通过静态分析辅助上下文复制实现内存隔离。在将 OmegaGen 移植到 C/C++ 系统时，我们将探索更有效的解决方案，例如写时复制。 (4) OmegaGen 生成看门狗来报告单个进程的故障。一项改进是将 OmegaGen 与故障检测器叠加层 [89] 配对，以便一个进程的故障检测器可以检查另一个进程的看门狗。 (5) 我们的看门狗目前专注于故障检测和定位，而不是恢复。我们将整合微重启 [58] 和 ROC 技术 [87]。

8. 相关工作

部分故障已在多种情况下进行了讨论。 ArpaciDusseau 和 Arpaci-Dusseau 提出了 fail-stutter 故障模型 [56]。普拉巴卡兰等人分析了磁盘的故障部分模型[88]。科雷亚等人提出了 ASC 故障模型[62]。黄等人提出了云中灰色故障的定义[76]。古纳维等人[68] 研究了硬件中的慢速性能故障。我们在第 2 节中介绍了侧重于现代云软件中的部分故障研究，而最近的一

项工作分析了由网络分区引起的云系统故障 [54]，我们的研究工作范围在其过程层面，即 网络分区可能会导致 分区进程 完全故障（与其他进程断开连接）。此外，我们的研究工作涵盖了网络问题之外的更多样化的故障根源。

故障检测已被广泛研究 [53, 59, 60, 63, 65, 71, 72, 80 - 82, 91]。但他们主要专注于检测分布式系统中的停止运行故障；部分故障超出了这些检测器的范围。 Panorama [75] 建议利用系统中的可观察性来检测灰色故障 [76]。虽然这种方法可以增强故障检测，但它假设一些外部组件能碰巧观察到细微的故障行为，这些逻辑观察组件也无法确认故障过程的哪一部分是有问题的，从而使后续的故障诊断非常耗时[32]。

看门狗定时器是嵌入式系统中必不可少的硬件组件[57]。对于通用软件，由于代码库规模大且程序逻辑复杂，手动构建看门狗更具挑战性。因此，使用看门狗概念的现有软件 [4, 14] 仅将看门狗设计为浅层健康检查（例如，http 测试）和终止策略 [42]。我们论文的立场[83] 是提倡内在的看门狗抽象并阐明其设计原则。OmegaGen 能够通过静态分析为给定程序自动生成全面的定制看门狗。

一些工作旨在生成软件不变量或简化运行时检查。 Daikon [64] 从动态执行跟踪中推断出可能的程序不变量。 PCHECK [92] 使用程序切片来提取配置检查，以检测初始化期间的潜在错误配置。 OmegaGen 则是对这些尝试的补充，我们专注于合成检查器，通过使用一种新颖的程序缩减技术来监控生产中程序的长期运行过程。

9．结论

系统软件持续地变得越来越复杂。这会导致现有解决方案无法捕捉到各种部分故障。这项工作首先对流行系统软件中的 100 个真实世界部分故障进行了研究，以阐明此类故障的特征。然后我们介绍 OmegaGen，它采用程序缩减方法来生成用于检测和定位部分故障的看门狗。随后在六个大型系统上评估 OmegaGen，它可以为每个系统生成数十到数百个定制的看门狗。结果是 生成的看门狗检测到 22 个实际部分故障中的 20 个，中位检测时间为 4.2 秒，并确定 18 个案例的故障范围；这些结果明显优于基准检测器。我们的看门狗还在最新的 ZooKeeper 中揭露了一个新的部分故障。

致谢

我们要感谢 NSDI 的审稿人和我们的指导人 Aurojit Panda，他们提出的宝贵意见使论文得到了改进。我们感谢 Ziyan Wang 为 OmegaGen 实现 wd_assert API 支持。我们感谢 Azure、AWS 和谷歌云平台慷慨的云研究信用支持。这项工作部分得到了 NSF 赠款 CNS-1755737 和 CNS-1910133 的资助。

参考

[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 W ALs 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] Y arn-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 V erità, 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 the1st 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 A TC’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. Y uan, 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, V erification 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, andR. 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, March2014.

[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 A TC ’ 18, page 387 – 399, Boston, MA, USA, 2018.

[90] R. V allée-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. Y uan, 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.

附录 A

惰性复制下的一致性 第 4.7 节描述了我们将上下文与三个属性（版本、弱引用和哈希）相关联，以处理由于惰性复制优化导致的潜在不一致性。在这里，我们给出一个具体的例子来阐明潜在的不一致是如何产生、以及如何解决它。使用惰性复制（本质上是“copy-on-get”），在上下文设置器中调用之后，上下文可能会被修改甚至失效；如果发生这种情况，getter 将复制不同的上下文值。例如，在检查器中调用上下文获取器时，oa 可能已经失效（垃圾收集）。但是由于 getter 会检查 weak_ref 属性，它会发现上下文无效（weak_ref 返回 null），因此不会复制。如果 oa 仍然有效，上下文获取器将进一步检查当前值的哈希码，如果与记录的哈希不匹配则跳过复制。这种方法是轻量级的。但它假设 Java 对象的哈希码约定在程序中得到遵守。如果不是这种情况，例如，不管其内容如何，oa 的哈希码都是相同的，则可能会出现不一致（getter 复制修改的上下文）。这种不一致可能会、也可能不会对检查器造成问题。对于上面的例子，检查器的写入，可能会写入“xxxtest”而不是“xxx”到看门狗测试文件，这仍然可以。但是如果另一个易受攻击的操作在“xxx”上有一个特殊的不变量，这种不一致会导致运行时的误报。我们在 12 小时实验期间的低误报率表明，哈希码约定违规通常不是成熟软件的主要问题。

```
        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);
}
```

另一个需要考虑的一致性场景，是当检查器使用一些需要多个上下文参数的易受攻击操作时。由于在惰性复制优化下，上下文检索是异步的，因此在 getter 检索所有参数时 可能会发生竞争条件。例如，在 getter 检索到 oa 之后，第二个参数（节点）在 getter 检索它之前更新。在这种情况下，两个参数都是有效的、并且匹配它们所记录的哈希属性。但是，它们是由 foo()

的两次调用混合而成。我们使用版本属性解决了这种不一致的情况。检查器在调用被检查操作之前会比较它需要的所有上下文的版本属性是否相同，如果版本不一致则跳过检查。

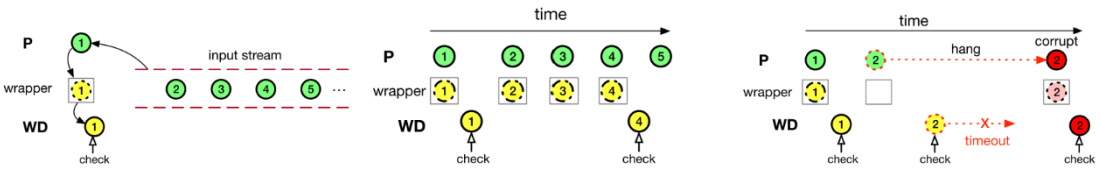
```
          Main Program                          Watchdog Checker
-----------------------------------   --------------------------

// 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);
}
```

附录 B 实现细节

幂等包装器 在 4.7 节我们描述了幂等包装器机制，它允许看门狗安全地调用非幂等操作，尤其是读取类型的操作。我们在此处进一步详细说明此机制的细节。

其基本思想是让看门狗和主程序以协调的方式调用包装器而不是原始操作。包装器将区分调用是来自主程序还是看门狗，以易受攻击的操作 readRecord 为例。在无故障场景下，主程序正常执行实际的 readRecord；看门狗检查器将获得一个缓存值。在故障场景下，主程序可能会卡在 readRecord 中；看门狗将被阻止在包装器的关键部分之外，因此它可以在不执行实际 readRecord 的情况下检测挂起。图 7 说明了这两种情况。



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

Figure 7: Illustration of idempotent wrapper

图 7：幂等包装器示意图： (a) readRecord 的幂等包装器 (b) 在正常情况下调用包装器 (c) 在错误场景中调用包装器

OmegaGen 会自动为所有读取类型的易受攻击操作 生成 幂等包装器。 OmegaGen 首先在主程序中定位所有调用读操作的语句，它从这些语句中提取流对象，为每种类型的流对象生成一个

包装器。看门狗驱动程序则维护流对象和包装器实例之间的映射。为了让包装器稍后执行实际操作，OmegaGen 为流类中的每个读取类型方法分配一个不同的操作编号，并根据操作编号生成一个调用该方法的调度程序。然后，OmegaGen 将原始调用替换为对看门狗驱动程序的包装入口点的调用，使用流对象、操作号和调用者源作为参数。例如，buf = istream.read(); 在主程序中将替换为 buf = WatchdogDriver.readHelp(istream, 1, 0); 其中 1 是读取的操作数，0 表示从主程序调用包装器。

读取类型操作的检查器构造中，剩余的其他步骤与其他类型易受攻击操作的步骤相类似。关键区别在于 OmegaGen 将为 包装操作 而不是 操作本身 生成一个独立的检查器。特别是，OmegaGen 生成的检查器将包含一个调用指令，以使用源 1（来自看门狗）作为参数来调用正确的包装器。

附录 C 评估补充

语义检查 API 我们在第 6 节中的实验中没有使用语义检查 wd_assert（第 4.5 节）来避免有偏差的结果。但我们确实在案例 ZK3 上使用 wd_assert 进行了测试。虽然看门狗 OmegaGen 会自动生成并检测到这种情况，但这是因为故障触发条件（坏磁盘）也影响了看门狗中其他一些易受攻击的 I/O 操作。我们编写了一个 wd_assert 来检查磁盘上的事务记录是否远远落后于内存中的记录：

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

OmegaGen 通过自动提取必要的上下文、封装看门狗检查器并从主程序中删除这个代价昂贵的语句来处理繁琐细节。生成的语义检查器可以在 2 秒内检测到故障并查明问题。

I/O 使用开销 在第 6.7 节中的开销实验相同的设置下，我们测量了 有 和 没有 看门狗的六个系统的磁盘 I/O 使用（使用 iotop）和网络 I/O 使用（使用 nethogs）。表 11 显示了结果，我们可以看到看门狗导致的 I/O 使用增加很小（磁盘 I/O 的中位数为 1.6%，网络 I/O 的中位数为 4.4%）。

iotop 是一款开源、免费的用来监控磁盘 I/O 使用状况的类似 top 命令的工具，iotop 可以监控进程的 I/O 信息。

nethogs 是一种开源的命令行工具(类似于 Linux 的 top 命令)，用来按进程或程序实时统计网络带宽使用率。

		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.

表 11：基本系统 和 带看门狗系统 的平均磁盘和网络 I/O 使用情况

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

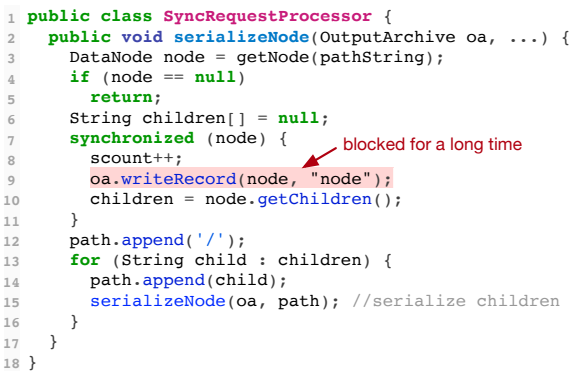
It is elusive to build large software that never fails. Designers of robust systems therefore must devise runtime mechanisms that proactively check whether a program is still functioning properly, and react if not. Many of these mechanisms are built with a simple assumption that when a program fails,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

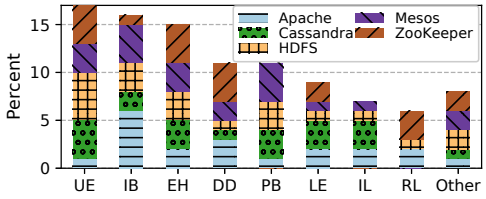


Figure 2: Root cause distribution. *UE*: uncaught error; *IB*: indefinite blocking; *EH*: buggy error handling; *DD*: deadlock; *PB*: performance bug; *LE*: logic error; *IL*: infinite loop; *RL*: resource leak.

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.

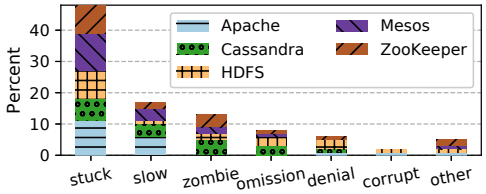


Figure 3: Consequence of studied failures.

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.

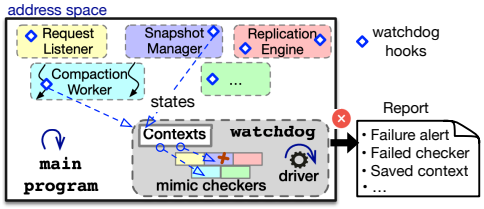


Figure 4: An intrinsic watchdog example.

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).

(a) A module in main program

(b) Generated checker

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

A complication arises when a method has multiple callsites, 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` ↪ `g` ↪ `h` in the main program, the reduced call chain is `f'` ↪ `g'` ↪ `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.

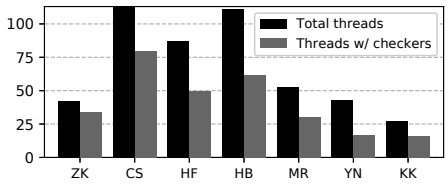


Figure 6: Thread-level coverage by generated watchdog checkers.

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×) 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);
}
```

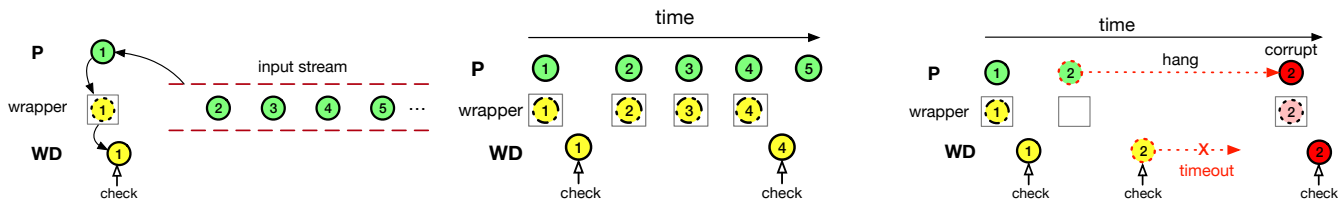
Id.	Root Cause	Conseq.	Sticky?	Study?
ZK1 [45]	Bad Synch.	Stuck	No	Yes
ZK2 [66]	Uncaught Error	Zombie	Yes	Yes
ZK3 [47]	Logic Error	Inconsist.	Yes	No
ZK4 [48]	Resource Leak	Slow	Yes	Yes
CS1 [7]	Uncaught Error	Zombie	Yes	Yes
CS2 [8]	Indefinite Blocking	Stuck	No	Yes
CS3 [12]	Resource Leak	Slow	Yes	No
CS4 [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).