

ENHANCING CLOUD SYSTEM RUNTIME TO ADDRESS COMPLEX FAILURES

by

Chang Lou

A dissertation submitted to Johns Hopkins University
in conformity with the requirements for the degree of
Doctor of Philosophy

Baltimore, Maryland

August, 2023

© 2023 Chang Lou

All rights reserved

Abstract

As the reliance on cloud systems intensifies in our progressively digital world, understanding and reinforcing their reliability becomes more crucial than ever. Despite impressive advancements in augmenting the resilience of cloud systems, the growing incidence of complex failures now poses a substantial challenge to the availability of these systems. With cloud systems continuing to scale and increase in complexity, failures not only become more elusive to detect but can also lead to more catastrophic consequences. Such failures question the foundational premises of conventional fault-tolerance designs, necessitating the creation of novel system designs to counteract them.

This dissertation aims to enhance distributed systems' capabilities to detect, localize, and react to complex failures at runtime. To this end, this dissertation makes contributions to address three emerging categories of failures in cloud systems. The first part delves into the investigation of partial failures, introducing OmegaGen, a tool adept at generating tailored checkers for detecting and localizing such failures. The second part grapples with silent semantic failures prevalent in cloud systems, showcasing our study findings, and introducing Oathkeeper, a tool that leverages past failures to infer rules and expose these silent issues. The third part explores solutions to slow failures via RESIN, a framework specifically designed to detect, diagnose,

and mitigate memory leaks in cloud-scale infrastructures, developed in collaboration with Microsoft Azure. The dissertation concludes by offering insights into future directions for the construction of reliable cloud systems.

Keywords: cloud-scale systems, distributed systems, software reliability, highly available systems, fault tolerance, failure detection, failure diagnosis, failure mitigation, program analysis, runtime verification.

Thesis Committee

Primary Readers

Ryan (Peng) Huang (Advisor)
Associate Professor
Department of Computer Science and Engineering
University of Michigan

Scott Smith
Professor
Department of Computer Science
Johns Hopkins University

Ding Yuan
Associate Professor
Department of Computer Science
University of Toronto

Suman Nath
Partner Research Manager
Microsoft Research Redmond

Acknowledgments

Foremost, I would like to express my deepest gratitude to my advisor, Prof. Ryan Huang. Ryan is the best Ph.D. advisor I could ever ask for. "Your Students Are Your Legacy", a well-known article from David Patterson, shares his guiding principles for mentoring Ph.D. students, emphasizing that faculties' lasting impact lies not in their published work, but in the people they nurture. Ryan is an advisor who wholeheartedly embodies this philosophy in his mentorship. Over the past six years, I have been incredibly fortunate to work alongside such a mentor who genuinely cares for mentees. His constant support making my journey far from a "lonely march on Mars." His persistence, critical thinking, attention to detail, and humility have profoundly influenced my personal growth. His encouragement, in the toughest times, has kept me going. I will treasure the moments we spent together at the milk tea shop after paper rejections, when he cheered us up with his favorite quote: "great work will eventually shine." As I move forward, I aim to mirror his focus on student well-being and growth.

I would also like to extend my sincere thanks to the members of my thesis committee, Prof. Scott Smith, Prof. Ding Yuan, and Dr. Suman Nath, for their unwavering support throughout my doctoral studies. It is a great privilege to have even one seasoned and empathetic advocate, so I consider myself extremely fortunate to

have the backing of three such individuals. Prof. Smith has been approachable and compassionate since my first day at Hopkins, always willing to lend a sympathetic ear. Prof. Yuan has contributed significantly to my job search through his invaluable advice. Dr. Nath has consistently provided insightful guidance that has effectively steered the direction of our work. Their genuine concern for my success has been a constant reminder of the impact a dedicated educator can have.

I appreciate the assistance I have received from other faculty members I encountered on this journey. I am grateful to the support within the Johns Hopkins Computer Science Department, especially from Prof. Soudeh Ghorbani and Prof. Yinzhi Cao. They spent valuable time to help rehearsal my job talk. I am thankful that Prof. Randal Burns, Prof. Amitabh Basu, Prof. Rebecca Schulman, and Prof. Nicholas Durr served in my GBO committee. I also want to thank external perceptive advice from Prof. Zaoxing Liu, Prof. Cheng Tan, and Prof. Tianyin Xu.

I would like to thank my mentors and collaborators from Microsoft Azure. Dr. Yingnong Dang has consistently encouraged and pushed me to strive for higher standards, and I am truly grateful for his guidance. My mentor Cong Chen wholeheartedly supported me during my multiple internships, even when I messed things up. After our memory leak detection tool ironically caused memory leaks on Azure production clusters, he spent several nights to help me with the issue, and tried to cheer me up, “at least the tool detected its own leaking.”

My research in computer systems would not have commenced without the help of Prof. Haibo Chen and Prof. Rong Chen from Shanghai Jiao Tong University. They guided me through the fog after my first failed graduate school application attempt and mentored my first system project. Their recommendations were crucial to lead me to work with Ryan. I also wish to express my gratitude to Prof. Junfeng

Yang, his student Rui Gu from Columbia University, and Zhenyu Guo from Microsoft Research, who demonstrated the excitement of system research and led me to this career path. Early in my career, I benefited significantly from the advice of senior researchers, notably Prof. Shan Lu and Prof. Cheng Li.

To my fellow Computer Science Department friends, thank you for your companionship throughout this journey. It has been a long road, and I am grateful that I never had to walk it alone. Starting this journey alongside my lab mate Yigong, spending countless nights running experiments, and dreaming of our accomplishments together was a joy. Being his witness at the wedding was an honor I hold dear. I received much help from senior Ph.D. students when starting: Kuan, Zeyu, Kunal, Disa, and James. I am lucky to start the graduate school with many other friends who are great inspirations: Hang, Zhihao, Zhuolong, Zhen, Zhengzhong, Yu, Cong, Siyuan, Zhuotun, Jingfeng, Shiwei, Megna and many others. I extend my best wishes to other OrderLab members: Brian, Haoze, Yuzhuo, and Diwang, wishing you all have successful careers.

I want to thank amazing intern students I have worked with: Xu, Ziyang, Ding, Dimas, Zhewen, and Yujin. Our research projects could not be done with their dedication and efforts.

My Ph.D. studies was made possible by the unwavering motivation, inspiration, and support of my family. I am deeply grateful to my parents, Yexin and Biyun, for their unceasing care despite our geographic separation. Their love is a constant source of strength for me. To my partner, Jenny, thank you for your patience and understanding through these many years. Your presence in my life is a miracle, and I eagerly anticipate our future adventures together.

To people and things that I never lose affection

Previously Published Material

Chapter 2 incorporates contents from a previous publication [199]: Chang Lou, Peng Huang, Scott Smith. Comprehensive and Efficient Runtime Checking in System Software through Watchdogs. In Proceedings of the 17th Workshop on Hot Topics in Operating Systems, Bertinoro, Italy, May 2019. HotOS’19.

Chapter 3 revises a previous publication [151]: Chang Lou, Peng Huang, Scott Smith. Understanding, Detecting and Localizing Partial Failures in Large System Software. In Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation, Santa Clara, CA, February 2020. NSDI’20.

Chapter 4 revises a previous publication [153]: Chang Lou, Yuzhuo Jing, Peng Huang. Demystifying and Checking Silent Semantic Violations in Large Distributed Systems. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, CA, July 2022. OSDI’22.

Chapter 5 revises a previous publication [154]: Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, Murali Chintalapati. RESIN: A Holistic Service for Dealing with Memory Leaks in Production Cloud Infrastructure. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, CA, July 2022. OSDI’22.

Table of Contents

Table of Contents	x
List of Tables	xvi
List of Figures	xviii
1 Introduction	1
1.1 The Rise of Complex Failures	1
1.2 Advancing Runtime for Complex Failures	3
1.3 Dissertation Contributions	4
1.4 Dissertation Outline	6
2 Background	7
2.1 Fault-tolerance in Distributed Systems	7
2.1.1 Fail-stop	8
2.1.2 Byzantine Faults	9
2.2 Challenges of Dealing with Complex Failures	10
2.3 Conclusion	13
3 Partial Failures	14

3.1	Introduction	14
3.2	Real-world Failure Study	19
3.2.1	Findings	20
3.2.2	Implications	24
3.3	Watchdog: An Intrinsic Failure Detector Abstraction	25
3.4	OmegaGen: Automatically Generate Watchdogs	29
3.4.1	Identify Long-running Methods	30
3.4.2	Locate Vulnerable Operations	31
3.4.3	Reduce Main Program	33
3.4.4	Encapsulate Reduced Program	34
3.4.5	Add Checks to Catch Faults	35
3.4.6	Validate Impact of Caught Faults	36
3.4.7	Prevent Side Effects	37
3.5	Implementation	42
3.6	Evaluation	43
3.6.1	Generating Watchdogs	43
3.6.2	Detecting Real-world Partial Failures	45
3.6.3	Localizing Partial Failure	49
3.6.4	Fault-Injection Tests	51
3.6.5	Discovering A New Partial Failure Bug	51
3.6.6	Side Effects and False Alarms	52
3.6.7	Performance and Overhead	53
3.6.8	Sensitivity	55
3.6.9	Semantic Check API	55
3.7	Limitations	56

3.8	Conclusion	56
4	Silent Semantic Failures	58
4.1	Introduction	58
4.2	Background	62
4.2.1	Definition	62
4.2.2	An Example	63
4.3	Real-world Failure Study	64
4.3.1	Study Methodology	64
4.3.2	Prevalance	66
4.3.3	Violated Semantics	70
4.3.3.1	Sources	70
4.3.3.2	Categorizations	72
4.3.4	Root Causes	76
4.3.5	Manifestations	77
4.3.6	Current Practice	80
4.3.6.1	Testing	80
4.3.6.2	Assertions	81
4.3.6.3	Observability	82
4.4	Oathkeeper: Checking Semantic Violations	83
4.4.1	Design Overview and Workflow	83
4.4.2	Instrumentation and Trace Generation	86
4.4.3	Template-Driven Inference	88
4.4.4	Rule Validation	91
4.4.5	Runtime Checking	93

4.4.6	Optimizations	93
4.4.7	Implementation	94
4.4.8	Limitations	95
4.5	Evaluation	95
4.5.1	Generation Overview	95
4.5.2	Checking Newer Violations	96
4.5.3	Performance	98
4.5.4	Runtime Overhead	99
4.5.5	Rule Activation and False Positive	99
4.6	Conclusion	100
5	Memory Leaks at Cloud-scale	103
5.1	Introduction	103
5.2	Background and Motivation	107
5.2.1	Host Memory Compositions	107
5.2.2	Memory Leaks	108
5.2.3	Requirements	110
5.3	RESIN: Automating Resolving Memory Leaks at Cloud-scale . . .	111
5.3.1	Overview	111
5.3.2	Design of Leak Detection	113
5.3.2.1	Challenges	114
5.3.2.2	Lightweight Memory Usage Monitoring	114
5.3.2.3	Detection Algorithms	116
5.3.3	Diagnosis of Detected Leaks	122
5.3.3.1	Background: Heap Snapshot	122

5.3.3.2	Choosing Candidate Hosts to Profile	124
5.3.3.3	Deciding Trace Collection Strategy	124
5.3.3.4	Collecting Reference Snapshots	127
5.3.3.5	Trace Analyses for Diagnosis	128
5.3.4	Mitigating Leaks	129
5.4	Evaluation	131
5.4.1	Deployment Status and Scale	131
5.4.2	Detecting Production Memory Leaks	132
5.4.3	End-to-End Impact	133
5.4.4	Effectiveness of Detection	133
5.4.5	Effectiveness of Diagnosis	135
5.4.6	Effectiveness of Mitigation	139
5.4.7	Comparison of Different Algorithms	139
5.4.8	Runtime Overhead	141
5.4.9	Tuning Effort	143
5.5	Lessons and Limitations	144
5.6	Conclusion	145
6	Related Work	146
6.1	Distributed Systems Failure Study	146
6.2	Failure Detection	147
6.3	Failure Diagnosis	147
6.4	Runtime Verification	148
6.5	Distributed Tracing and Monitoring	149

7	Conclusion	150
7.1	Limitations of Our Approach	150
7.2	Lessons for Robust Cloud System Designs	151
7.3	Future Directions	152
7.4	Concluding Remarks	154

List of Tables

2.1	Code base sizes of popular distributed systems in their recent releases.	12
3.1	Studied software systems, the partial failure cases, and the unique versions, version and date ranges these cases cover.	19
3.2	Evaluated system software.	43
3.3	Number of watchdogs and checkers generated.	43
3.4	22 real-world partial failures reproduced for evaluation.	45
3.5	Four types of baseline detectors we implemented.	46
3.6	Detection times (in seconds) for the real-world cases in Table 3.4. .	47
3.7	Failure localization for the real-world cases in Table 3.4.	47
3.8	False alarm ratios (%) of all detectors in the evaluated six systems.	52
3.9	OmegaGen watchdog generation time (sec).	53
3.10	System throughput (op/s) w/ different detectors.	53
3.11	Average disk and network I/O usages of the base systems and w/ watchdogs.	54
4.1	Number of public interfaces in popular distributed systems. An interface can have multiple semantics under different settings.	59

4.2	Studied systems, the tickets (of various kinds) in the issue tracker of each system, the cases we sampled, and cases studied.	65
4.3	Some templates integrated in Oathkeeper.	87
4.4	Evaluated newer semantic failures.	102
4.5	System throughput (op/s) with varying percentages of semantic rules enabled.	102
5.1	Leak patterns, characteristics and their completion triggers.	126
5.2	Single mitigation action execution time (seconds).	139
5.3	VM deployment time (seconds) impact by trace collection.	142

List of Figures

2.1	Traditional failure detection models in distributed systems.	8
3.1	A production ZooKeeper outage due to partial failure [175].	16
3.2	Root cause distribution.	21
3.3	Consequence of studied failures.	22
3.4	An intrinsic watchdog example.	27
3.5	Example of watchdog checker OmegaGen generated for a module in ZooKeeper.	28
3.6	Illustration of idempotent wrapper	41
3.7	Thread-level coverage by generated watchdog checkers.	44
4.1	A silent semantic failure in ZooKeeper.	64
4.2	Issue priorities of semantic failure cases and all valid sampled cases.	67
4.3	Consequence of the studied semantic failures.	67
4.4	Sources of violated semantics.	68
4.5	Issue ticket age (creation date minus first release date).	72
4.6	Timing of semantic violation.	77
4.7	Distribution (# of cases) of the failure triggering conditions. Some combinations are omitted in the diagram for readability.	79

4.8	Workflow of Oathkeeper.	85
4.9	Inference and validation algorithm example.	91
4.10	Example: Oathkeeper workflow of using ZK-1208 to detect ZK-1496.	94
4.11	Detection of 22 semantic failures in ZooKeeper (sorted by the bug ticket time in ascending order) when applying Oathkeeper on a slid- ing subset of the failures for inferring semantic rules.	97
4.12	Time to generate trace, infer rules, and verify rules against test suite.	98
5.1	A production memory leak example in Azure from a host process that caused leaks of objects allocated at the kernel side.	109
5.2	Workflow of the RESIN system.	113
5.3	Monitor agent in each node collecting memory usage data.	115
5.4	The memory usage grouped into buckets and pivot by image name, service, and bucket size.	118
5.5	Applying the moving suspicious interval algorithm.	121
5.6	Periodic heap snapshot collection.	123
5.7	Memory growth patterns and completion point choices. Each data point is real memory usage from production processes.	125
5.8	Memory leak cases RESIN detected and reported.	131
5.9	Unexpected VM reboot.	132
5.10	VM allocation error rate.	132
5.11	Ticket resolved and w/ trace ratio.	132
5.12	The fix for ServiceH leak.	137
5.13	Contract violation induced leak on ServiceD.	137
5.14	Mitigation for a leaking driver.	138

5.15 False positive of detection algo.	138
5.16 False negative of detection algo.	138
5.17 Ranks of root cause stack trace in diagnosis analyses on 8 trace files.	141
5.18 Memory size and CPU usage changes through tracing.	143

Chapter 1

Introduction

1.1 The Rise of Complex Failures

Cloud services play an increasingly essential role in our daily lives. They have re-defined traditional business models [4], emerged as the primary mechanism for processing data workloads [47], as well as profoundly reshaping social and entertainment experiences [124]. The convenience and efficiency offered by these services have revolutionized the way we work and interact.

Failures within cloud systems are, unfortunately, an inescapable fact. In 2022 alone, Google Cloud documented over 400 failure incidents [76]. These failures often lead to large-scale service unavailability and substantial economic losses [83, 10, 210, 170, 73]. Lloyd's of London has forecasted that a solitary cloud incident, specifically involving the shutdown of a top U.S. cloud computing provider, could potentially trigger close to \$19 billion in business losses [187]. Evidently, real-world instances like the Amazon Web Services outage in 2020, which disrupted a significant portion of the internet including services like Roku, Adobe, and Flickr, underline this point. Failures not only disrupt the immediate service but also erode user trust, posing a long-term threat to the sustainability of cloud services.

Historically, many strategies have improved system availability and cloud resilience [41, 8, 205, 27, 42, 185, 3, 56, 45, 63, 88], including consensus protocols like Paxos [136] for reliable data agreement and RAID [182] for data redundancy amid disk failures. Fault-tolerant mechanisms, such as checkpointing [123], allow quick system recovery and minimal downtime. Collectively, these efforts have resulted in cloud systems achieving 99.9% availability, an industry benchmark [24, 11, 169]. There have also been significant advancements in bug finding [157, 126], system testing [145, 140], tracing [203, 23, 64, 190, 159, 131], and safe deployment [146, 225], further fortifying the resilience of these systems.

Nevertheless, the increasing complexity of cloud systems is evident in the failures developers face. Cloud systems rapidly evolve in scale and complexity. Microsoft plans to built up to 100 new data centers every year [171]. Amazon Web Services announced it will invest \$7.8 billion in new data centers [22]. As cloud systems introduce new features such as serverless computing, adopt new architectural paradigms like microservices, and enforce security and compliance regulations such as the General Data Protection Regulation [207], they become unprecedentedly complex.

Consequently, both industry practitioners and research communities have observed a rise of “complex failures” [121, 82, 156, 222, 226, 118] – *a state that systems appear functional while being defective*. Complex failures occur across different stacks, from software to hardware [20, 200], caused by diverse factors such as misconfiguration [194], OS scheduling [195], bad disk [21], faulty network device [193], etc. Previously dismissed as glitches or corner cases, these failures now hold significant costs. For instance, a bad configuration caused a 59-minute incident in 2021, leading to a \$34 million revenue loss for Amazon [12]. These issues can no longer be ignored.

1.2 Advancing Runtime for Complex Failures

A direct strategy to decrease complex failures would be bug exposure through program analysis [113, 212, 219] or testing [220, 196, 225]. However, bugs in the production software are inevitable. Even with extensive testing and strict code reviews, bugs may still escape and manifest in the production. In practice, failures often have diverse root causes, making it challenging to employ a common pattern for bug detection. Many failures require specific production workloads or timing to trigger as shown in studies [151], rendering most of them production-dependent and necessitating runtime mechanisms to handle.

Existing runtime approaches, unfortunately, prove insufficient in dealing with complex failures. Today, cloud systems have multiple defenses against failures, while their effectiveness relies on efficient failure detection, a process that monitors nodes' states and recognizes failures. Successful detection is the key to successful recovery. Existing mechanisms show competence in managing crash failures [42, 204, 2, 25, 94, 45], but they fall short in detecting complex failures. We have observed cases including gray failures [121], partial disk failures [183], limplock [60], fail-slow hardware [17, 82], metastable failures [119], and state corruption [55], all of which are prevalent in large cloud infrastructures.

This dissertation confronts a key question: can we enhance cloud system runtime to improve availability [1] amidst escalating failure complexities? To address this, our research initiates with a series of studies to thoroughly understand complex failures in production environments, using empirical observations to both deepen our comprehension and guide our system design. Subsequently, we reimagine traditional

failure detection with new failure detector abstractions tailored to system logic, aiming to capture and localize obscure failures that may have otherwise escaped the radar. Meanwhile, constructing fine-grained detectors for large and complex cloud systems is not easy. Towards this goal we construct comprehensive and robust solutions by integrating different techniques from programming languages, software engineering, and machine learning. Specifically, our solution often includes an automated tool to extract insights from programs, tests, and execution traces. Such tool can eliminate the need for developers to write formal specifications or implement additional checkers. With this approach we can synthesize fine-grained failure detectors for large systems by utilizing derived insights.

Thesis statement *The increasingly complex and subtle failures present a formidable threat to the health of modern cloud systems. Enhancing system capabilities to detect, localize, and react to complex failures is crucial to improve the cloud system availability. Leveraging techniques like program analysis and machine learning offers scalable solutions for such large and intricate systems.*

1.3 Dissertation Contributions

This dissertation made contributions to three emerging types of cloud failures.

Firstly, we identified partial failures, where a process remains active but loses some of its functionalities, as a significant issue for many production systems. We noticed these failures often led to catastrophic consequences, like data loss and inconsistencies, and were difficult to detect and debug. We concluded that the root problem was the inability of existing failure detectors to comprehend the complex

failure semantics of modern systems. To counter this, we proposed intrinsic software watchdogs, which closely observe anomalies within a process. We automated the creation of these watchdogs using a tool we developed, called OmegaGen, which successfully generated hundreds of watchdogs for six popular distributed systems, detecting and localizing a majority of partial failures.

Secondly, our work on partial failures inspired our discovery on silent semantic violations, a type of failures subtly violate system semantics without giving off explicit error signals. These violations can lead to long-lasting damages and incorrectness. To understand this problem, we conducted an empirical study on 109 real-world silent semantic failures from nine widely-used distributed systems. Based on our findings, we built Oathkeeper, a tool that automatically infers semantic rules from past failures and enforces them at runtime to detect new failures. Oathkeeper managed to generate a large number of rules for the evaluated systems and proved effective at detecting new violations with minimal overhead.

Finally, we explored solutions for slow failures, notoriously known for their lack of definitive standards for detection. We tackled the common slow failure example of memory leaks on the cloud scale, which had been an arduous problem for cloud developers/operators. We identified the unique characteristics of the cloud environment that complicate the handling process after closely observing developers' process in fixing memory leak bugs. To address the issue, we collaborated with Azure engineers and developed RESIN, a centralized service that analyses processes across all hosts in the Azure fleet to detect, diagnose, and mitigate memory leaks. Since its implementation, RESIN has substantially reduced the number of VM reboots caused by low memory across Microsoft Azure.

1.4 Dissertation Outline

The structure of this dissertation is as follows:

Chapter 2 provides an overview of the fundamental concepts about failure detection in distributed systems. It elaborates on traditional failure detection mechanisms and their limitations, underscoring the need for new solutions.

Chapter 3 delves into our work concerning partial failures. This includes an empirical study on failures from widely-used distributed systems and the design and assessment of our solution, OmegaGen. This chapter forms the foundation of this dissertation.

Chapter 4 sheds light on our research into silent semantic failures. Our primary focus lies in the thorough analysis of silent semantic failures in production. Additionally, we expound on the design and evaluation of our solution, Oathkeeper.

Chapter 5 outlines our collaborative work on cloud memory leaks with Azure, and introduces the design and evaluation of the RESIN system. Please note, certain confidential details have been anonymized for privacy reasons.

Chapter 6 presents an extensive review of previous literature, approaches, and techniques in the field of distributed systems, especially focusing on related fields such as failure studies, failure detection, runtime verification.

Chapter 7 discusses the limitations of our approach, the lessons we learned from our research and outlines exciting future directions that this field of research may explore.

Chapter 2

Background

This chapter provides a brief review of the existing literature in the domain of distributed system failure detection. It underlines the the conventional mechanisms of failure detection, and highlights the shortcomings of the existing models, setting the stage for the contribution of this thesis.

2.1 Fault-tolerance in Distributed Systems

A distributed systems is an ensemble of independent computing entities. Each server, often referred to as node, is interconnected via a network, which collaborate to perform complex tasks. Developers build a variety of services on top of this model such as storage [54, 30, 69, 58], processing [57, 221] and coordination [29].

Reliability is a paramount requirement for distributed services. Many services provide mission-critical operations in industries such as finance, healthcare, and telecommunications. Reliable distributed systems must ensure uninterrupted service availability, maintain data consistency across various nodes, and promptly recover from unexpected failures. These characteristics directly influence user experience and trust, and the performance of businesses that rely on these services.

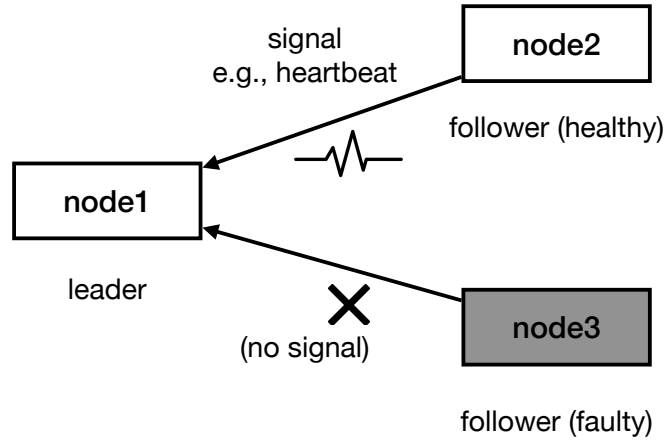


Figure 2.1: Traditional failure detection models in distributed systems.

In the context of distributed systems, failure refers to the inability of a system or its component to perform its intended function. Failures can be attributed to a multitude of reasons, including hardware faults, network partitioning, software bugs, or operator errors. Reliable system designs must consider potential failures.

The reliability requirement necessitates fault tolerance, a process that enables a system to continue to perform (at certain level) even if a failure happens in software or hardware. This is achieved primarily through mechanisms such as redundancy, where multiple instances of the same service are available to handle requests. For example, if a single node fails in a distributed cluster, the system can redirect tasks to another operational node, thereby ensuring continuity of service.

In the subsequent portion of this section, we will discuss two principal fault tolerance models: *Fail-stop* and *Byzantine Faults*.

2.1.1 Fail-stop

The fail-stop model is a simplified failure model used in the design and analysis of distributed systems. In this model, components are abstracted as uniformed nodes,

allowing group protocols to manage inherent complexities. Components need to detect problematic peers that have potentially crashed, commonly through timeouts and heartbeat mechanisms. As shown in Figure 2.1, components frequently transmit heartbeat signals, indicating their operational status and enabling prompt responses to failures. The system operates under the assumption that components are either functioning or non-functioning.

Due to its simplicity and efficiency, the fail-stop model has gained popularity in practice. Numerous research [42, 185, 45, 63, 94] studies and aims to enhance its accuracy, primarily focusing on differentiating true crash failures from delayed response due to network delays or congestion [42, 3]. Chandra and Toueg [42] introduced the concept of unreliable failure detectors and showed Consensus can still be solved even with unreliable failures detectors despite mistakes. GossipFD [56] and SWIM [56] propose scalable protocols to detect crash failures in large-scale systems.

However, this failure mode is too simple to represent the complex behavior of modern software. Its binary assumption of component status becomes insufficient as cloud failures grow increasingly complex. In reality, it is possible for a component to *appear functional while being defective*: system processes continue to run while internal threads have stopped, or a storage service fail to replicate the promised number of data copies. Even a simple firewall service can gradually consume memory resources, potentially bringing down the host server.

2.1.2 Byzantine Faults

A Byzantine Fault, as introduced by Leslie Lamport, Robert Shostak, and Marshall Pease in their 1982 "Byzantine Generals Problem" [137], is an error in a distributed network, where a component behaves unpredictably. This analogy represents faulty

network nodes behaving like traitorous generals, spreading conflicting information, leading to confusion and a lack of consensus. These faults are difficult to detect due to their inconsistent symptoms and their potential to spread seemingly true, yet false information. This unpredictable nature separates Byzantine Faults from other, more predictable system failures.

To counteract such faults, an extensive array of Byzantine Fault Tolerance (BFT) [41, 134, 114, 206] techniques was introduced. BFT enables systems to operate correctly and reach consensus despite some components acting arbitrarily or maliciously, effectively mitigating the problems illustrated by the Byzantine Generals Problem. BFT algorithms aim to reach agreement on a single data value among nodes. Traditional BFT algorithms tolerate up to one-third faulty nodes.

However, BFT is not a panacea for all failure modes. For instance, it is ill-equipped to handle gradual performance decay [49], such as a node's processing capacity deteriorating due to hardware issues in a distributed cloud service. Furthermore, BFT is an expensive solution, often reserved for security-critical applications like blockchains. Consequently, a more efficient approach is desirable.

2.2 Challenges of Dealing with Complex Failures

We have discussed why existing failure detectors prove to be inadequate to handle emerging failures in production: the fail-stop model, which treats all monitored software as coarse-grained nodes, is overly generic. On the other hand, BFT is unnecessarily expensive.

The fundamental problem with existing failure detectors is, their logic is disjoint from monitored system execution: it merely embeds a loop that periodically sends

“ping” to peers, which cannot represent the complex behavior of modern software. Thus, such coarse-grained detectors will create a false sense of health even when the main program is broken for a long time or it will “bark” excessively even when the main program executes smoothly.

In contrast, good failure detectors should intersect the normal execution so they accurately reflect the *internal* status of the main program. They should be tailored to different behaviors in different system components, and check safety and liveness requirements these components may potentially violate, e.g., checking if a Cassandra background task of SSTable compaction is stuck. Detectors should also achieve high coverage of the main program’s major components.

Crucially, detectors should also provide hints for recovery options. Many cloud failures are extremely difficult to diagnose and can take days or weeks to pinpoint [129]. To expedite recovery, besides enhance the detection of ongoing failures, detectors should also pinpoint the problematic code region along with the payload for diagnosing and reproducing production failures.

Challenge 1: Design new abstractions for failure detectors, specifically tailored to accommodate system logics for detecting and localizing complex failures.

Some researchers recognized the need for new fault tolerance techniques to address increasingly complex failure semantics. For example, Falcon [141] employs developer-written checking functions to differentiate potential non-responsiveness causes. For ZooKeeper, the checking function tests whether a client request has been processed recently, while concurrently, a separate component issues no-operation requests at a minimal rate. This manual approach,

Software	SLOC (approx.)	Packages	Classes	Methods
Zookeeper	28,000	16	192	3,562
Cassandra	102,000	58	826	12,919
HDFS	219,000	110	1431	79,584
HBase	160,000	70	962	17,868

Table 2.1: Code base sizes of popular distributed systems in their recent releases.

however, proves impractical for larger, more realistic settings. Modern cloud infrastructures typically contain hundreds of diverse services, ranging from data storage and processing to network management and security. These services often operate under different paradigms and have extensive codebases, as depicted in Table 2.1, with many surpassing hundreds of thousands of lines. This increasing size exacerbates the complexity of monitoring and error detection tasks.

Given the extensive nature of cloud system codebases, it’s unrealistic to expect developers to manually create checkers. This process is not only time-consuming, but also susceptible to human error. Consequently, automation becomes imperative, as it can substantially reduce both the time expenditure and the likelihood of mistakes. Any effective solution should accommodate large-scale programs, leading us to:

Challenge 2: Propose principles to automate failure detector construction for systems with substantial codebases.

Detectors should not only be effective, but also safe. Ideally, detectors should operate as mere “observers”, minimizing interference with program execution. However, this requirement contradicts with the need to accurately reflect a program’s status, which requires detectors closely monitor program execution by inserting “probes” and code snippets to frequently synchronize and utilize state with the main program. These hooking or probing actions may inadvertently disrupt the program state or

cause synchronization issues, leading to inaccurate results or crashes. They may also introduce performance overheads that are detrimental in performance-sensitive environments.

Challenge 3: Prevent side effects caused by failure detection, including correctness and performance issues, while maintain detection accuracy.

2.3 Conclusion

This chapter highlighted the critical role of fault tolerance, the types and implications of failures, and the fundamental aspects of failure detection. It also pointed out the existing gaps in current failure detection methods, justifying the need for the research presented in this thesis. The subsequent chapters delve into the proposed solution to address these gaps for different types of failures and improve the efficiency and effectiveness of failure detection in distributed systems.

Chapter 3

Partial Failures

3.1 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 [167, 166, 9, 78, 175, 70, 71, 74, 6]. 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 [167].

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 [133], e.g., `create()` requests time out but `write()` requests still work. In a production ZooKeeper outage due to the leader failing partially [175], 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 3.1). 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 [94], but still conclude that its current design “*has very little ability to effectively do something non-trivial to deal with partial failures*” [40]. Prabhakaran *et al.* analyze partial failure specific to disks [183]. Huang *et al.* discuss the gray failure [121] challenge in cloud infrastructure. The overall characteristics of software partial failures, however, are not well understood.

```

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 }

```

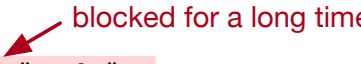


Figure 3.1: A production ZooKeeper outage due to partial failure [175].

In this chapter, 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 3.2) of 100 real-world partial failure cases from five large-scale, open-source systems. We find that nearly half (48%) of the studied failures 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 [198, 14]). 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 [120], 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 3.1 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 3.1 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

Software	Lang.	Cases	Vers (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 3.1: Studied software systems, the partial failure cases, and the unique versions, version and date ranges these cases cover.

cases. Through testing, our watchdogs exposed a new, confirmed partial failure bug in the latest ZooKeeper.

3.2 Real-world Failure Study

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 [86]. Researchers studied partial disk faults [183] and slow hardware faults [82]. 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 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 3.1). 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).

3.2.1 Findings

Finding 1: *In all the five systems, partial failures appear throughout release history (Table 3.1). 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 [112] 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 [110].

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 3.2).*

Results are shown in Figure 3.2 (*UE*: uncaught error; *IB*: indefinite blocking; *EH*:

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

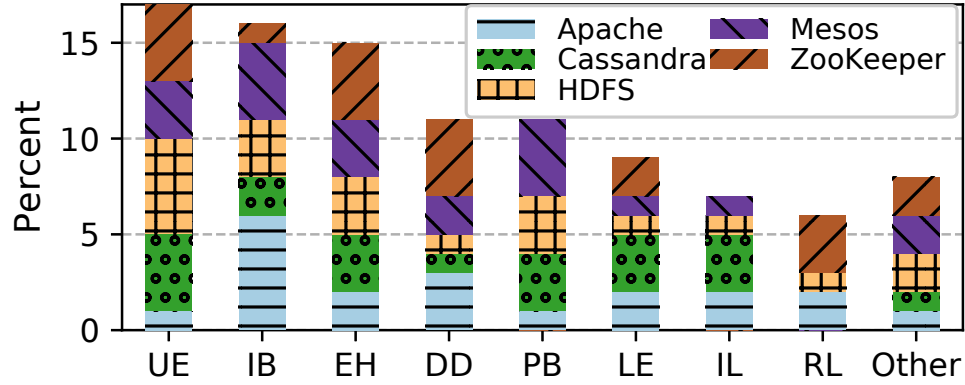


Figure 3.2: Root cause distribution.

buggy error handling; *DD*: deadlock; *PB*: performance bug; *LE*: logic error; *IL*: infinite loop; *RL*: resource leak.) 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` [33]. Indefinite blocking occurs when some function call is blocked forever. In one case [108], 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 [219], premature continuing, *etc.* Other common root causes include deadlock, performance bugs, infinite loop and logic errors.

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

Figure 3.3 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

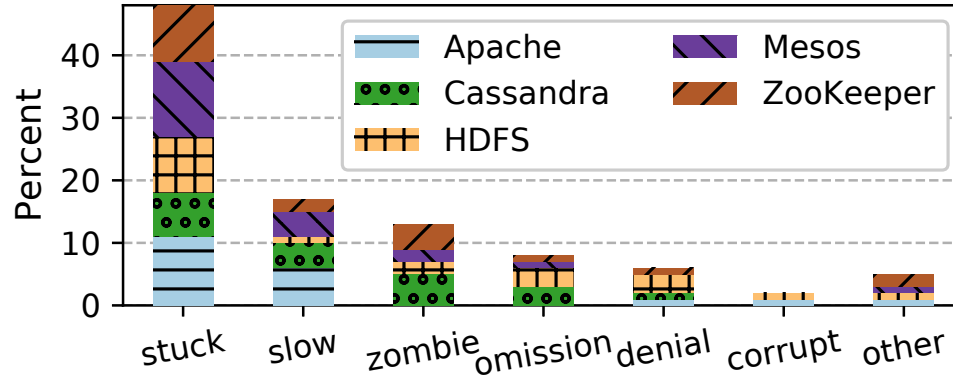


Figure 3.3: Consequence of studied failures.

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 3.3). 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 [32], 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 [229]. In another case, the HDFS datanode continued even if the block pool failed to initialize [107], 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 [165]. In another case [172], 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 [70]. Another partial failure in the ZooKeeper leader would only occur when a connecting follower hangs [235], 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 [36]. This thread must be restarted to fix the issue. For certain failures, some extra repair actions such as fixing a file system inconsistency [104] 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 [108]. However, these non-sticky failures already incurred damage for a long time by then (15 minutes in one case [228]).

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

For example, diagnosing a Cassandra failure [37] 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.

3.2.2 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 [85], or observers [120] 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 [108]. 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.

3.3 Watchdog: An Intrinsic Failure Detector Abstraction

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 3.4) 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.

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

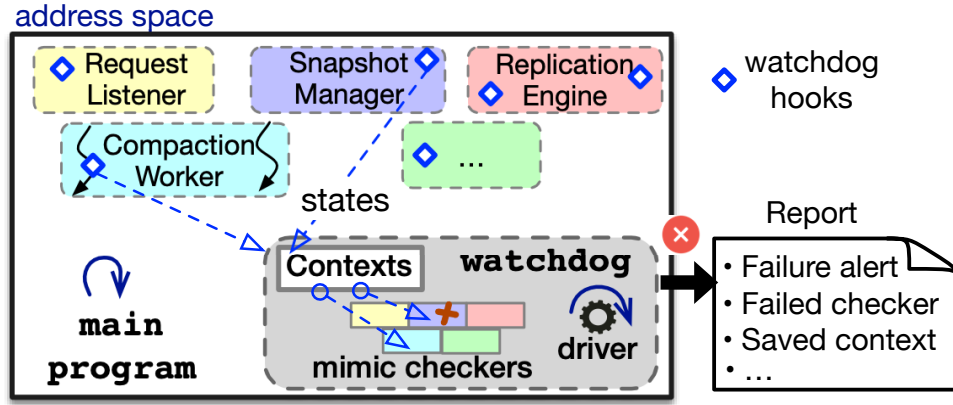


Figure 3.4: An intrinsic watchdog example.

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 [14]) 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 (§3.2). 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

```

1 public class SyncRequestProcessor {
2     public void run() {
3         while (running) { ① identify long-running region
4             if (logCount > (snapCount / 2))
5                 zks.takeSnapshot();
6             ...
7         } ③ reduce
8     }
9 }
10 public class DataTree { ③ reduce
11     public void serializeNode(OutputArchive oa, ...) {
12         ...
13         String children[] = null;
14         synchronized (node) { ② locate vulnerable operations
15             scount++;
16             oa.writeRecord(node, "node");
17             children = node.getChildren();
18         }
19         ...
20     } + ContextManger.serializeNode_reduced
21     _args_getter(oa, node);
22     ④ insert context hooks

```

(a) A module in main program

```

1 public class SyncRequestProcessor$Checker {
2     public static void serializeNode_reduced(
3         OutputArchive arg0, DataNode arg1) {
4         arg0.writeRecord(arg1, "node");
5     }
6     public static void serializeNode_invoke() {
7         Context ctx = ContextManger. ④ generate
8         serializeNode_reduced_context(); context
9         if (ctx.status == READY) { factory
10             OutputArchive arg0 = ctx.args_getter(0);
11             DataNode arg1 = ctx.args_getter(1);
12             serializeNode_reduced(arg0, arg1);
13         }
14     }
15     public static void takeSnapshot_reduced() {
16         serializeList_invoke();
17         serializeNode_invoke();
18     }
19     public static Status checkTargetFunction0() {
20         ... ⑤ add fault signal checks
21         takeSnapshot_reduced();
22     }
23 }

```

(b) Generated checker

Figure 3.5: Example of watchdog checker OmegaGen generated for a module in ZooKeeper.

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 [160]. 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.

3.4 OmegaGen: Automatically Generate Watchdogs

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 3.3. 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 3.5 shows an overview example of running OmegaGen.

As discussed in Section 3.2.2, 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).

3.4.1 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 (§3.4.4).

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

predicate is a runtime property associated with a method which tracks whether a call site of this method is in fact reached.

For an invocation target inside a potentially long-running loop, a hook is inserted before the loop that sets its predicate and another hook after the loop that unsets its predicate. A callee of a potentially long-running method will have a predicate set to be equal to this caller’s predicate. At runtime, the predicates are assigned and evaluated that activates or deactivates the associated watchdog. The predicate instrumentation occurs after OmegaGen finishes the vulnerable operation analysis (§3.4.2) and program reduction (§3.4.3).

3.4.2 Locate Vulnerable Operations

OmegaGen then analyzes the identified long-running methods and further narrows down the checking target candidates (step ②). 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 3.5a 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 [81] – 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 [70], 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.

3.4.3 Reduce Main Program

With the identified long-running methods and vulnerable operations, OmegaGen performs a top-down program reduction (step ③) starting from the entry point of long-running methods. For example, in Figure 3.5a, OmegaGen will try to reduce the `takeSnapshot` function first. When walking the control flow graph of a method to be reduced, if an instruction is tagged as potentially vulnerable, it would be retained in the reduced method. Otherwise, it would be excluded. For a call instruction that is not tagged as vulnerable yet, it would be temporarily retained and OmegaGen will recursively try to reduce the target function. If eventually the body of a reduced method is empty, *i.e.*, no vulnerable operation exists, it will be discarded. Any call instructions that call this discarded method and were temporarily retained are also discarded.

The resulting reduced program not only contains all vulnerable operations reachable from long-running methods but also preserves the original structure, *i.e.*, for a call chain $f \hookrightarrow g \hookrightarrow h$ in the main program, the reduced call chain is $f' \hookrightarrow g' \hookrightarrow h'$. This structure can help localize a reported issue. In addition, when later a watchdog invokes a validator (§3.4.6), the structure provides information on which validator to invoke.

If a type of vulnerable operation (*e.g.*, the `writeRecord` call in Figure 3.5a) 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.

3.4.4 Encapsulate Reduced Program

OmegaGen will encapsulate the code snippets retained after step ③ 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 3.5a 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 ④). 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 (§3.4.1). 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 3.5a, 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.

3.4.5 Add Checks to Catch Faults

After step ④, 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 [141] 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 (§3.2). 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 (§3.4.4) 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.

3.4.6 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 (§3.4.3) 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.

3.4.7 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* [122, 115] 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. 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
<pre> ----- void foo() { foo_reduced_args_setter(oa); write(oa); oa.append("test"); <--- oa = foo_reduced_ctx.args_getter(0); } </pre>	<pre> ----- void foo_reduced_invoke() { </pre>

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
<hr style="border-top: 1px dashed orange;"/>	
<i>// called in a loop</i>	
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.

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.

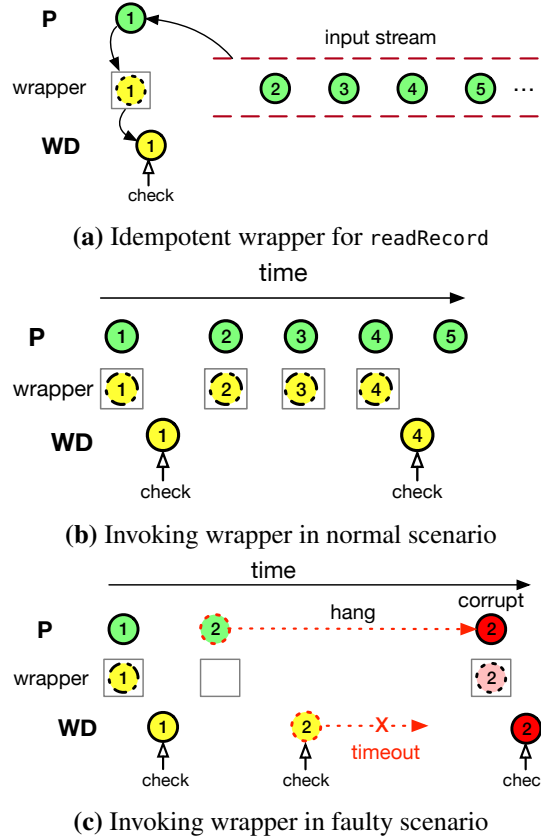


Figure 3.6: Illustration of idempotent wrapper

With this approach, 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 as 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 3.6 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. In particular, the checker OmegaGen generates will contain a call instruction to the proper wrapper using source 1 (from watchdog) as the argument.

3.5 Implementation

We implemented OmegaGen in Java with 8,100 SLOC. Its core components are built on top of the Soot [202] program analysis framework, so it supports systems in Java bytecode. OmegaGen does not rely on specific JDK features. The Soot version we

	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 3.2: Evaluated system software.

	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 3.3: Number of watchdogs and checkers generated.

used can analyze bytecode up to Java 8. We leverage a cloning library [135] 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 watchdogs. A single script automates the workflow and packages the watchdogs with the main program into a bundle.

3.6 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.

3.6.1 Generating Watchdogs

To evaluate whether our proposed technique can work for real-world software, we evaluated OmegaGen on six large systems (Table 3.2): *ZK* (ZooKeeper), *CS* (Cassandra), *HF* (HDFS), *HB* (HBase), *MR* (MapReduce), *YN* (Yarn). We chose these

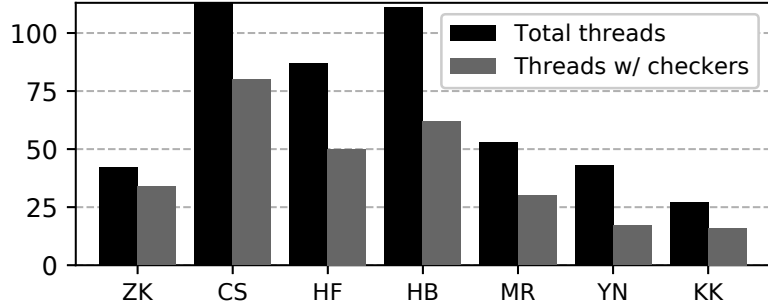


Figure 3.7: Thread-level coverage by generated watchdog checkers.

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 3.3 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 (§3.4.1). 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 3.7 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 (§3.4.3).

Id.	Root Cause	Conseq.	Sticky?	Study?
ZK1 [228]	Bad Synch.	Stuck	No	Yes
ZK2 [70]	Uncaught Error	Zombie	Yes	Yes
ZK3 [230]	Logic Error	Inconsist.	Yes	No
ZK4 [232]	Resource Leak	Slow	Yes	Yes
CS1 [34]	Uncaught Error	Zombie	Yes	Yes
CS2 [35]	Indefinite Blocking	Stuck	No	Yes
CS3 [39]	Resource Leak	Slow	Yes	No
CS4 [38]	Performance Bug	Slow	Yes	No
HF1 [110]	Uncaught Error	Stuck	Yes	Yes
HF2 [103]	Indefinite Blocking	Stuck	No	Yes
HF3 [102]	Deadlock	Stuck	Yes	No
HF4 [109]	Uncaught Error	Data Loss	Yes	No
HB1 [99]	Infinite Loop	Stuck	Yes	No
HB2 [97]	Deadlock	Stuck	Yes	No
HB3 [101]	Logic Error	Stuck	Yes	No
HB4 [100]	Uncaught Error	Denial	Yes	No
HB5 [96]	Indefinite Blocking	Silent	Yes	No
MR1 [163]	Deadlock	Stuck	Yes	No
MR2 [162]	Infinite Loop	Stuck	Yes	No
MR3 [164]	Improper Err Handling	Stuck	Yes	No
MR4 [161]	Uncaught Error	Zombie	Yes	No
YN1 [215]	Improper Err Handling	Stuck	Yes	No

Table 3.4: 22 real-world partial failures reproduced for evaluation.

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 3.4 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 one week on average to reproduce each failure. Seven cases are from our study in Section 3.2. Others are new cases we did not study before.

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

Detector	Description
Client (Panorama [120])	instrument and monitor client responses
Probe (Falcon [141])	daemon thread in the process that periodically invokes internal functions with synthetic requests
Signal	script that scans logs and checks JMX [178] metrics
Resource	daemon thread that monitors memory usage, disk and I/O health, and active thread count

Table 3.5: Four types of baseline detectors we implemented.

comparison (Table 3.5). The client checker is based on the observers in state-of-the-art work, Panorama [120]. The probe checker presents Falcon [141] app spies (which are also manually written in the Falcon paper). When implementing the signal and resource checkers, we follow the current best practices [198, 52] and monitor signals recommended by practitioners [116, 188, 201, 13].

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 [116].

Result Table 3.6 shows the results. ✖ means the failure is undetected. 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

	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 3.6: Detection times (in seconds) for the real-world cases in Table 3.4.

	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	●	●	●
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
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

Table 3.7: Failure localization for the real-world cases in Table 3.4.

errors.

In comparison, as Table 3.6 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 (§3.6.6).

Case Studies ZK1 [228]: This is the running example in 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 3.1). OmegaGen generates a checker `serializeNode_reduced`, which exposed the issue in 4 s.

CS1 [34]: 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 [96]: 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 [38]: 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 [215]: 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.

3.6.3 Localizing Partial Failure

Detection is only the first step. We further evaluate the localization effectiveness for the detected cases in Table 3.6. 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 3.7 shows the result. ➡ means pinpointing the faulty instr. * means pinpointing the faulty function or data structure. ✱ means pinpointing a func in the faulty function’s call chain. ▶ means pinpointing some entry function in the

program, which is distant from the root cause. ● means only pinpointing the faulty process. ○ means misleadingly pinpointing another innocent process. N/a meanings not applicable because failure is undetected. 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 [163], 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 [103], 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 [100], 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.

3.6.4 Fault-Injection Tests

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

3.6.5 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 [234], which

	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 3.8: False alarm ratios (%) of all detectors in the evaluated six systems.

has been confirmed by the developers and fixed.

3.6.6 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 (§3.4.7), 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 (§3.6.7); *loaded*: random node restarts, every 3 minutes into the moderate workloads, switch to aggressive workloads ($3\times$ number of clients and $5\times$ request sizes); *tolerable*: run with injected transient errors tolerable by the system. Table 3.8 shows the results. Each cell reports the ratio range under three setups (stable, loaded, tolerable). *watch_v* means watchdog with validators. 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.

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

Table 3.9: 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 3.10: System throughput (op/s) w/ different detectors.

These false alarms would disappear once the transient faults are gone. With the validator mechanism (§3.4.6), 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%).

3.6.7 Performance and Overhead

We first measure the performance of OmegaGen’s static analysis. Table 3.9 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 [59] with 15 clients sending 15,000 requests (40% read); for Cassandra, we used YCSB [53] with 40 clients sending 100,000 requests (50% read); for HDFS, we used built-in benchmark NN Bench Without MR 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 4.5 shows that the watchdogs incur 5.0%–6.6% overhead on throughput.

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

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.

We measured the disk I/O usages (using `iostat`) 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 3.6.7. Table 3.11 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).

3.6.8 Sensitivity

We evaluate the sensitivity of our default 4-sec timeout threshold on detecting liveness issues with ZK1 [228] (stuck failure) and ZK4[232] (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 (§3.4.5), 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.

3.6.9 Semantic Check API

Our experiments in Section 3.6 did not use semantic checks, `wd_assert` (§3.4.5), 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.

3.7 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 [197] 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 [31] and ROC techniques [180].

3.8 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.

Chapter 4

Silent Semantic Failures

4.1 Introduction

Users' increasing reliance on distributed systems highlights the importance of ensuring they work correctly. Unfortunately, real-world distributed systems inevitably encounter failures. When a failure is recognizable through explicit signals such as crash, timeout, error code, or exception, timely actions can still be taken to detect [42, 141, 151] and mitigate [181, 179, 144] the failure. A vexing problem occurs when a system is operational but *silently* breaks its semantics without apparent anomalies.

Take a distributed notification service as an example, which provides an interface that promises to invoke the client callback whenever the status of some object changes. A bug may cause this system to miss invoking the callback upon a change or invoke the callback more than necessary. As another example, a distributed file system that is supposed to replicate data blocks by user-configured n copies may incorrectly under-replicate some blocks without any explicit errors.

Such failures can lead to severe consequences because they violate the guarantees a system provides to its users. They also break the contracts that other components or applications rely on, and result in amplified incorrectness. Moreover, since the

System	Ver.	Client API	Public Method	Admin Command	Config.
ZooKeeper	3.4.6	38	219	13	30
ZooKeeper	3.6.2	78	2,853	18	128
HDFS	2.7.2	128	5,293	11	224
HDFS	2.10.0	162	6,306	12	449
Kafka	2.6.0	166	2,661	76	366
Kafka	2.8.0	171	3,107	86	379

Table 4.1: Number of public interfaces in popular distributed systems. An interface can have multiple semantics under different settings.

violation is silent, the damage exacerbates over time. For example, as the buggy distributed file system that silently violates its replication policy continues to run, more and more newly created files will be subject to potential data loss.

Distributed systems today have rich semantics (Table 4.1) exposed through client APIs, public methods including RPCs among internal components, administrator commands, configuration parameters, *etc.* One interface often encodes multiple guarantees. New interfaces and semantics are also continuously introduced as a system evolves. These characteristics together make it challenging to ensure that a distributed system conforms to its semantics in production settings.

Indeed, real-world evidence shows that semantic violations occur in practice. In a Google cloud incident [77], a traffic engineering subsystem that is supposed to throttle traffic upon congestion incorrectly throttled traffic even though the network was not congested. Another highly-impactful global outage [75] was caused by a quota system incorrectly reporting the usage for a user ID service as zero.

However, other than anecdotal evidence, the problem of silent semantic violations in distributed systems remains mysterious, despite its severe consequences. For instance, mature distributed systems include extensive test cases to check the correctness of their features. Thus, it is natural to assume silent semantic violations are

rare in production because testing likely has eliminated most of them. In addition, while adding assertions and runtime verification [148, 147, 155, 216] are potential solutions, the conventional wisdom is that they are expensive and semantic rules are difficult to get. It is also unclear what kind of semantics are violated in practice.

To systematically understand this problem, we present, to our best knowledge, the first empirical study on 109 *real-world* silent semantic violations from nine widely-used distributed systems. Through these cases, we analyze key questions such as *how prevalent are semantic violations in practice, what semantics are violated, why are these failures not caught in testing, and how are these silent violations detected*.

Our study provides quantitative data points to answer these questions. The study findings also challenge some conventional wisdom and reveal gaps in the current practice. We highlight several findings:

- Contrary to the belief that silent semantic violations rarely occur in deployed systems, they have significant presence (39%) among sampled failures of all kinds.
- While the studied systems get more extensively tested over time and continue to add new features and semantics, their initial semantics do *not* become more bullet-proof. On the contrary, more than two thirds of the failures violate semantics that have existed since the system’s first stable release.
- Although these are distributed system failures, most (74%) violations can be determined locally in some component.
- The violated semantics are often *not* untested but rather well covered by existing test cases.
- Enabling assertions in release builds helps by converting semantic violations into crash failures. One studied system does this and has the lowest ratio of semantic failures.

- In many cases, although a semantic was initially honored, it was later violated, thus one-time assertions are insufficient.
- Many system semantics are vulnerable to violations during maintenance operations or node events.

Given the prevalence (as our study indicates) and severity of silent semantic violations, we design a tool Oathkeeper to help users check silent semantic violations at runtime. The tool design is directly guided by insights from our study.

Specifically, we find that in 73% of the cases, developers add regression tests after the failure is reported, which contain valuable information about the failed semantic. However, the majority of the studied cases still violate semantics that have been tested before. A major reason for the gap is that these regression tests are usually patch-driven: they only check if the specific bug is fixed in a particular setup using a bug-triggering workload. The underlying semantics can continue to be broken with different root causes in different scenarios.

Based on this insight, Oathkeeper leverages the regression tests and tries to infer the underlying semantic rules implied by the tests. To do so, Oathkeeper runs the tests on both the buggy version and patched version of the system, and takes a *template-driven* approach to automatically infer semantic rules from the two traces. Oathkeeper then deploys these semantic rules to production to catch future violations that are caused by different bugs under different conditions.

We evaluate Oathkeeper on ZooKeeper, HDFS, and Kafka. Oathkeeper infers hundreds to thousands of semantic rules from the old regression tests in these systems. With the inferred rules, we evaluate Oathkeeper on seven real-world semantic failures that were introduced long (9–34 months) after the old failures. Oathkeeper detects violations for six of them. With all rules enabled, Oathkeeper on average only

incurs 1.27% throughput overhead to the target systems.

The contributions of this chapter are two-fold: (i) the first study on *real-world* silent semantic violations in nine popular distributed systems; (ii) the design of Oathkeeper, which automatically infers semantic rules for large distributed systems to check silent semantic violations at runtime.

The source code of Oathkeeper is publicly available at:

<https://github.com/OrderLab/OathKeeper>

4.2 Background

4.2.1 Definition

We consider a distributed system \mathcal{S} that provides services through a collection of operations. Each operation o has certain *semantics* [72]. The semantics encode guarantees that o makes about the output, system states, and results of subsequent operations, in response to some triggering condition c . The condition c can be a client request, an admin command (at the server side), a message from internal components, as well as an environment change including the passage of time. The semantics of \mathcal{S} are all the guarantees provided by the history of operations \mathcal{S} executes in response to a list of c .

A semantic violation (failure) occurs when \mathcal{S} breaks some of its semantics in an execution. The failures may exhibit explicit error signals, such as crashes, timeouts, and exceptions. In such cases, the violations overlap largely with existing failure models and can be well addressed by existing techniques.

This work focuses on *silent* semantic violations, in which \mathcal{S} violates its semantics but remains operational without exhibiting explicit error signals (\mathcal{S} is unaware of its

misbehavior). We focus on this class of failures because they are under-studied yet incur damaging consequences, and they pose significant challenges to testing, failure detection, and recovery.

Silent semantic violations differ from other failure modes in observability. Fail-stop failures cause complete loss of functionality, which can be observed with simple measures such as monitoring heartbeats. Fail-slow [82], partial failures [151] and gray failures [121] only cause some functionality to be broken (slow). But these issues can still be observed with generic approaches, *e.g.*, checking exceptions or timeouts [150]. In comparison, silent semantic violations are difficult to observe without a deep understanding of \mathcal{S} ' semantics and execution.

Another way to interpret the “silent” aspect is on the semantics being violated. If \mathcal{S} only has a few operations, all of which have well-defined and thoroughly checked semantics, semantic violations in \mathcal{S} will be observable failures. Unfortunately, distributed systems have a large number of interfaces (Table 4.1), many of which have loosely-defined (or hidden) semantics that cannot be easily checked. Consequently, violations of such semantics are difficult to detect and address.

4.2.2 An Example

We show an example of silent semantic failures from our study (Section 4.3.1). ZooKeeper is a coordination service with a hierarchical data model. Its clients store data by creating `znode` in a namespace. A special type of `znode` is called ephemeral node. The semantics of the ephemeral node `create()` operation guarantees that the `znode` exists for as long as the creating client's session and will be deleted once the associated session ends. The triggering conditions are the `create` request and the client session disconnection. Ephemeral nodes are commonly used to store membership

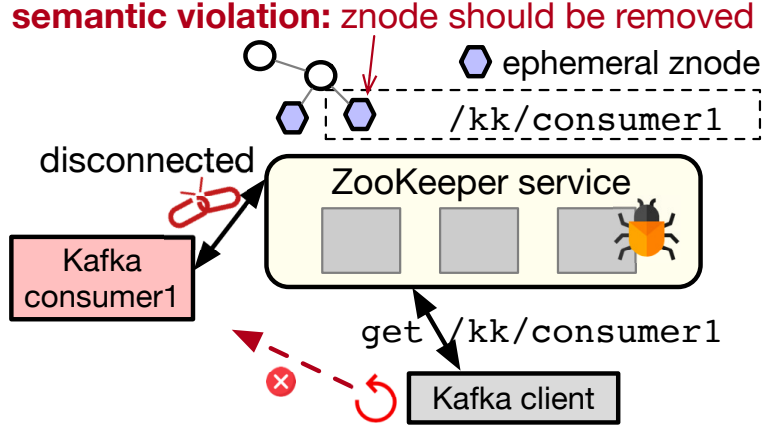


Figure 4.1: A silent semantic failure in ZooKeeper.

information. For example, HDFS implements its leader election using ephemeral nodes [65].

In a production ZooKeeper failure [227], some ephemeral node still existed even though the client session that created them was long gone. Specifically, a Kafka consumer crashed but the associated znode was not deleted (Figure 4.1). As a result, when Kafka clients queried ZooKeeper to discover consumer information, they kept trying to connect to the crashed consumer. In other settings, this semantic violation can propagate to other dependent applications, *e.g.*, it will break HDFS namenode’s automatic fail-over feature, which depends on the ephemeral node semantics, causing an HDFS service outage.

4.3 Real-world Failure Study

4.3.1 Study Methodology

Compared to other failure modes in distributed systems, silent semantic violations are not well understood. To fill this gap, we conduct a study on *user-reported* silent semantic failures from nine large-scale distributed systems (Table 4.2). We select

System	Category	Lang.	All	Sampled (valid)	Candi -date	Stud -ied
Cassandra (CS)	Database	Java	3,308	69 (54)	25	12
CephFS (CF)	File Sys.	C++	673	673 (123)	37	12
ElasticSearch (ES)	Search	Java	4,101	101 (46)	26	10
HBase (HB)	Database	Java	6,143	233 (80)	32	14
HDFS (HF)	File Sys.	Java	3,409	99 (52)	22	14
Kafka (KF)	Streaming	Scala	2,764	142 (92)	39	13
Mesos (ME)	Cluster Mgr.	C++	2,462	116 (47)	21	12
MongoDB (MG)	Database	C++	14,776	355 (151)	30	10
ZooKeeper (ZK)	Coordination	Java	1,141	134 (102)	36	12
Total			38,786	1,922 (747)	268	109

Table 4.2: Studied systems, the tickets (of various kinds) in the issue tracker of each system, the cases we sampled, and cases studied.

these systems because they are representative, mature, widely used in production, and record many user-reported failures.

To collect the failure cases, we first query the study systems’ issue trackers to find tickets that (1) are marked as “bugs”, (2) have priorities higher than “minor”, (3) are resolved, (4) involve the server components. This step returns a large number of tickets. We then *randomly* sample a subset (Table 4.2). Among this subset, some are not real failures, such as issues found in internal testing. The remaining ones (*valid* column in Table 4.2) are potential production failures. We then read their descriptions and check whether the failures violate system semantics. We filter crashes, aborts, out-of-memory errors, and semantic failures with clear error signals.

After the above step, we get a candidate set of production silent semantic failures (*Candidate* column). Due to time constraints, we perform in-depth analyses on a subset of the candidate cases, preferring those with sufficient information and discussions. This gives us the final study dataset (*Studied* column) of 109 production semantic failure cases.

Note that our sample sizes vary across systems. This is because the studied systems’ tickets vary greatly in terms of their information, quality, and bug types. If using a fixed sample size or ratio, one system can dominate the study and produce extremely biased findings. Our sampling instead is done iteratively: for a particular system, if after an initial sampling, its number of *Candidate* cases is too small or 0, we sample more, until the candidate numbers for different systems are relatively balanced. Note that each iteration in this process is still randomly choosing from the *All* tickets.

Threats to Validity. Like all empirical studies, our study is subject to validity problems such as the representativeness and biases. We cover popular distributed systems of different types, such as database, file system, and search engine, to improve the representativeness. To minimize selection bias, we *randomly* sample the cases. We also spread the sampling across times so we are not biased by some specific version. To reduce the manual inspection errors, we write a detailed analysis document for each case and have multiple inspectors examine each document to reach a consensus.

Although our study provides informative findings on semantic failures in the studied systems, they may not be generalized to other systems beyond the scope this study was conducted. Our study is also biased by programming languages (Java and C++); the findings may not generalize to systems written in other languages such as Erlang or Elixir, which embrace “let-it-crash” error handling philosophy [16].

4.3.2 Prevalence

Prevalence. An important question about silent semantic violations is whether they occur rarely in production. Getting accurate prevalence data requires examining

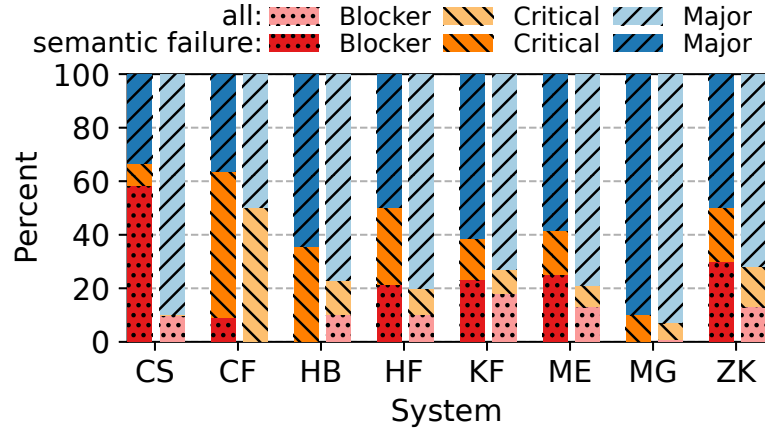


Figure 4.2: Issue priorities of semantic failure cases and all valid sampled cases.

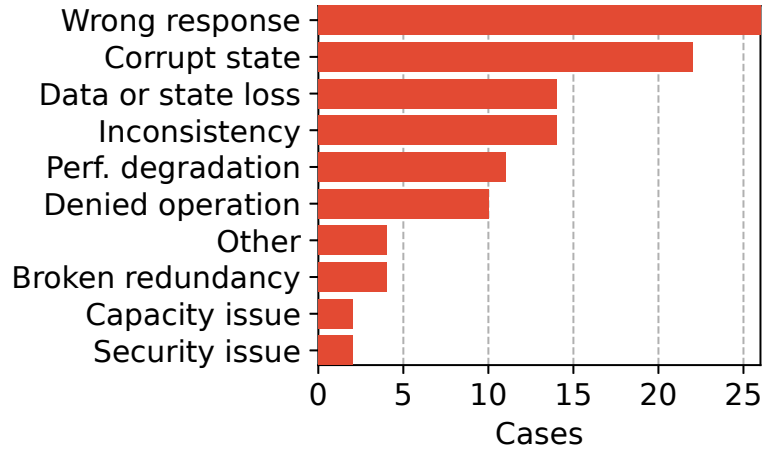


Figure 4.3: Consequence of the studied semantic failures.

thousands of tickets for each system, which is a daunting task. We instead obtain an approximate result by calculating the percentage of silent semantic failures in our sample set. Specifically, we calculate the percentages of the number of *candidate* cases in Table 4.2 over the number of *valid* cases in the sample. Note that the candidate cases are examined to be indeed silent semantic failures, even though we only study a subset of them.

Finding 1: *Silent semantic failures have significant presence across all studied systems, occupying 20%–57% (39% on average) of the sampled cases for all types of failures.*

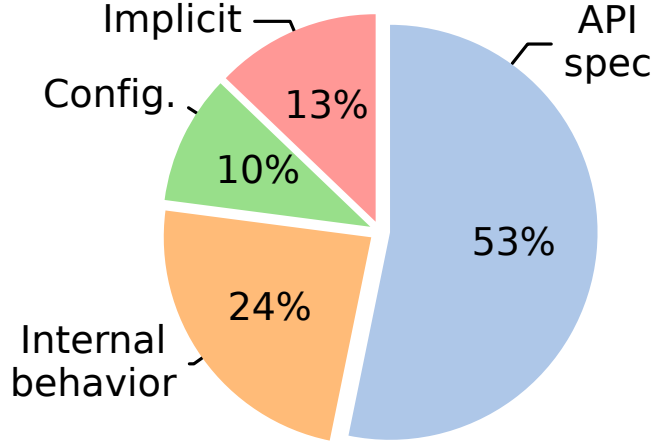


Figure 4.4: Sources of violated semantics.

The percentages vary in different systems. Systems such as ElasticSearch and Cassandra have a higher percentage of semantic failures (57% and 46%, respectively). MongoDB has the lowest ratio (20%). We will discuss in Section 4.3.6 these systems’ practices that may contribute to the differences.

Severity. How severe are these reported silent semantic failures? To answer this question, we analyze the severity levels that developers assign to the issues. Some systems use slightly different categories. We normalize them into three levels: *Blocker*, *Critical*, *Major*. Based on the official descriptions, *Blocker* means the issue “should block release until it is resolved”; *Critical* means the issue causes severe consequences like data loss; *Major* means a “major loss of function”.

Overall, 45% of the studied cases have *Blocker* or *Critical* priorities. The ZooKeeper failure [227] described in Section 4.2.2 is an example *Blocker* issue. As another example *Blocker* issue, users reported that in their HDFS deployment, all the replicas of

some blocks are residing on the same rack [111], which breaks the redundancy policy. This is clearly a severe violation because replica placement is critical to HDFS data.

We also compare the priority distribution of semantic failures with all failures in the sample. The result is shown in Figure 4.2. The average percentage of Blocker priority in semantic failures increases from 15% to 21%, and the percentage of Critical priority increases from 8% to 24%.

Interestingly, we find in some cases initially developers may not consider the symptoms to be severe, but after further investigation developers upgrade the priority level, *e.g.*, “*Marking as critical for 2.0. These ‘unexpected behaviors’ cause operator head-scratching and wasted hours of digging*” [98].

Finding 2: *Despite the lack of explicit error symptoms, silent semantic failures are considered severe by developers and users. Moreover, the sampled semantic failures are assigned with higher priorities compared to all sampled failures.*

Consequence. We next analyze the failure consequences. Figure 4.3 shows that besides incorrectness, semantic failures cause serious consequences such as corruption and data loss.

The consequences are damaging because clients or users are misled by the system’s seemingly normal reactions. For example, Kafka guarantees that when a success response is sent to a producer, the produced message will be persisted by at least `min.isr` replicas. Otherwise, the producer will be notified of an error, so it may retry the request. In one failure [130], a leader replica switched to follower then back to leader. Some messages produced were lost while the client received responses with no error. This false success resulted in data loss for the users.

Note that Figure 4.3 is about the reported impact of failures, which is not always the semantic violation per se. For example, in a MongoDB case, the maximum cache usage configuration is not enforced. It takes a while for the violation to cause a performance problem—which is the consequence of this failure. But even before the system reaches the performance collapse, a cache limit violation has occurred.

Finding 3: *In addition to incorrectness (wrong responses), silent semantic violations often cause severe consequences including corrupt state, data or state loss, and security issues.*

4.3.3 Violated Semantics

4.3.3.1 Sources

The studied failures violate various system-specific semantics. We analyze where these semantics come from. There are four sources and Figure 4.4 shows their distributions:

- **API spec:** a system API promises certain effect will (not) occur, *e.g.*, a successful return of `removeWatch` API is supposed to remove the specified watcher.
- **Internal behavior:** the system’s documentation explicitly guarantees that something should (not) occur about its *internal* behavior, which is not directly exposed to external APIs, *e.g.*, HDFS guarantees that if some Erasure Coding blocks fail, they should be detected and reconstructed.
- **User configuration:** user configurations regulate some system behaviors and the guarantees depend on the user settings. For example, the `max_hint_window_in_ms` parameter in Cassandra defines the maximum time window the coordinator will generate hints for a dead host.

- **Implicit:** the semantics are not explicitly defined or documented, but users expect them to hold for a correct system.

Finding 4: *Most (87%) studied failures violate semantics that are explicitly defined in API specs, system docs, or configs.*

Interestingly, in 10% of the studied cases, the system does not respect its configuration’s semantics. For example, if users set `acl.inheritance` to `true`, HDFS should enable ACL inheritance; but in one case the inherited ACL permissions are masked [106]. This violation causes security issues. The problem of misconfiguration is extensively researched [19, 18, 214, 117]. This finding suggests that even when users set configuration properly, a system can still misbehave.

As an example of *Implicit* semantics, in one HBase case [95], a region is online in server A, but the region location registered in the meta table is server B. While this consistency semantics is a common sense, it is not explicitly declared.

Explicit documentation of semantics is indicative of developers’ awareness of its guarantees and importance. One hypothesis is that if the semantics in a failure is not documented, it is understandable that developers did not make enough efforts to enforce the semantics. This finding disproves this hypothesis. However, the explicit documentations do *not* translate into fewer violations. One reason is that developers often document the semantics in a vague (e.g., “*should produce correct results*”) or incomplete way. A more fundamental gap is that the documentation is designed to be human-readable but not machine-checkable. For example, the semantics for ephemeral znode in ZooKeeper is documented clearly, but the system does not have any mechanism or tool to enforce this semantics in deployment.

Implications: *Rich sources of documentation exist to leverage and judge semantic*

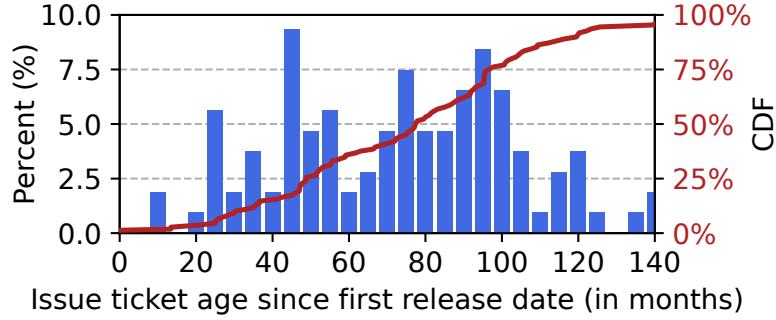


Figure 4.5: Issue ticket age (creation date minus first release date).

violations. Developers should move from documenting semantics in informal text to rigorously declare semantics that are mechanically checkable and enforceable.

4.3.3.2 Categorizations

Old vs. New Semantics Modern distributed systems often keep adding new features. For example, the number of client APIs in ZooKeeper increased from 38 in version 3.4.6 (2016) to 78 in version 3.6.2 (2020). Similarly, HDFS’ key APIs in `fs.FileSystem` increased from 128 in version 2.7.2 (2016) to 162 in version 2.10.0 (2019), along with significant increases of semantics in other interfaces such as RPC methods.

Since around 90% of our studied failures occurred after more than two years since the software’s initial release (Figure 4.5), a natural hypothesis is that most of them violate some new semantics. We validate this hypothesis by analyzing the age of semantics in the studied failures. We define *old semantics* as ones that exist since the first major stable release of the system and others as *new semantics*.

Surprisingly, we find only less than one third (32%) of our studied failures violate relatively new semantics, while 68% of them violate old semantics. Old semantics usually represent the most fundamental functionalities the system provides since developers implement them first, and they usually undergo extensive testing already.

However, our finding suggests that (1) even with new features added to the system, old semantics are still ones violated the most; (2) even with testing accumulating over the years, the reliability of old semantics is not necessarily higher in newer versions. Take ZooKeeper as an example. Its ephemeral znode interfaces and semantics have existed since the first major stable release (3.0.0) in October 2008 [15]. However, there are still production failures violating the guarantees of ephemeral znode reported by users even 10 years later [233].

We further investigate why old semantics still keep getting violated. There are three broad reasons: (1) *new implementation is buggy*, developers may optimize, refactor or refine the implementation of existing functionality, which contain bugs that break old semantics, *e.g.*, a concurrency bug introduced in changing an implementation to be multi-threaded; (2) *new feature adds buggy interactions*, when some new feature is added, developers may extend existing module to interact with or support the new feature. For example, after HDFS introduces the encryption zone feature, it needs to extend the original snapshot file function and the new handling path is buggy [105]; (3) *latent bugs are exposed*, as the most basic semantics, these old semantics' original implementations can be complex and contain latent bugs that can only be exposed in very specific scenario. In one ZooKeeper failure [231], users find the ephemeral znodes are not deleted when the system time changes unexpectedly. This bug exists for 6 years before it is discovered, because neither the testing nor most deployments would exercise the system with the time change.

Note that we did not count the numbers of semantics in the study, either for new or old semantics. This is because even with explicit documentation such as API specs, determining how many semantics are there for a given API can be subjective, which depends on the granularity of semantics. Instead, we objectively judge if the specific

semantics violated in a failure were introduced in the initial release or not.

Finding 5: *68% of the studied failures violate old semantics.*

Implications: *Instead of having the false hope that old semantics are reliable, developers should invest efforts to prevent semantic violation regressions.*

Local vs. Distributed Semantics Since the study subjects are distributed systems, we analyze whether the semantic violations naturally require considering multiple distributed components. This question is important to the design of runtime verification techniques [148, 147, 155, 216].

We find that indeed 26% of the semantic violations require global information to judge, *e.g.*, whether the replica placement policy in HDFS is correctly enforced, or whether states in different Cassandra nodes match the consistency level.

However, interestingly, we find that the majority (74%) of the violations can be determined in a local scope. For example, `appendTo` in HDFS has the semantics of appending data to the end of a target file and making it persistent. A buggy node may fail to persist the new blocks or accidentally overwrite them. The violations can be determined in this node.

One reason is that a distributed system component often keeps local copies of states for other components. For instance, even though ZooKeeper session is a global concept (a client connection to any follower or leader constitutes a session), such state is acknowledged to the ensemble. Thus, each node has a copy of the alive session and node list. The semantics of ephemeral `znode`, which require knowledge of the session information, can thus be checked locally in a ZooKeeper node.

Current runtime verification solutions typically aggregate global states across all

nodes to check property violations. Obtaining such global information can be both expensive and tricky, *e.g.*, dealing with consistency issues in capturing distributed snapshots [43]. Our finding suggests it may be sufficient to use local checkers to expose many semantic failures.

Finding 6: *The violations in semantic failures can be usually (74%) determined in the scope of a single component.*

Implications: *Employing local checkers can potentially expose many semantic violations.*

Safety vs. Liveness Semantics Some failures break safety-related guarantees. For example, in Kafka, the maximum number of consumers in a group should not be larger than a configured limit, but users found more consumers joined the group.

In comparison, other semantics are liveness related. For example, ZooKeeper specifies that a container-type znode with no child znodes should *eventually* be deleted. Even when we observe some empty container node exists, it does not necessarily indicate this guarantee is violated because it might still hold some time later. Without context, one can interpret some safety guarantee, such as a correct response should be returned, to be involving liveness, because even if a response is not received, it could be still on the way. We refer to the system’s official documentation for making the distinction. If the documentation explicitly states that when an operation returns, something (*e.g.*, a notification) will *eventually* happen, then a failure about its absence is a liveness violation.

It is generally challenging to check liveness properties [132], because there can be infinite possibilities in the execution that eventually produce the desired effect.

Fortunately, we find most (86%) of our studied failures violate safety semantics.

Finding 7: *86% of the studied cases violate safety semantics.*

Implications: *There is usually a fixed time point to determine if a system has violated its semantics.*

4.3.4 Root Causes

We analyze what causes a system to break its semantics. We are interested in identifying potential common bug patterns in the root causes, which can inform the designs of bug finding tools to eliminate semantic failures before production.

Some semantic failures are caused by bugs such as memory error, data race, and integer overflow, which are well studied with many tools designed to detect them. We find only 12% of the cases are caused by such bugs. The remaining failures are caused by system-specific logic bugs including design flaws, which are difficult to be caught by bug detection tools.

An interesting finding is that even for failure cases that violate the same or related semantics, their root causes can be quite different. Take the ZooKeeper ephemeral znode as an example: (1) ZK-1208 is caused by a race condition: when ZooKeeper is handling the close session request, it deletes ephemeral znodes and then removes the session, in between a create operation causes new ephemeral znodes to be added; (2) In ZK-3144, the violations are caused by an incorrect order: during request processing, the `lastProcessedZxid` is updated before sessions are modified, so a snapshot may not include the change and the ephemeral node is not deleted after log replay; (3) In ZK-2355, the violations are caused by buggy error handling: follower fails while reading the proposal packet, but resetting `lastProcessedZxid` is missed in the

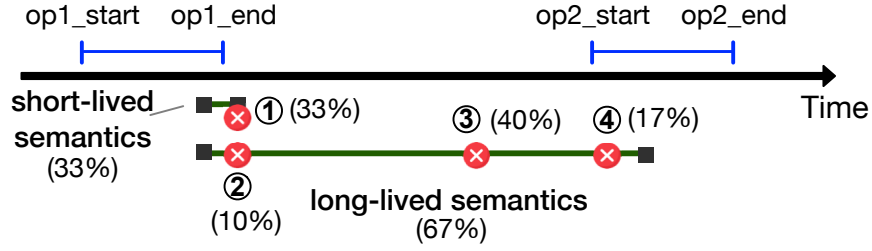


Figure 4.6: Timing of semantic violation.

error handler; (4) In ZK-2774, the system time of a server is changed unexpectedly, and session expiration codes rely the absolute system time, which causes the ephemeral znodes to persist after the client is disconnected for a long time.

Finding 8: *Only 12% of the studied failures are caused by well-defined bugs such as race conditions, while most cases are caused by a wide variety of logic bugs. Even for failures violating the same semantics, the root causes are diverse.*

Implications: *It can be challenging to exploit code patterns to eliminate semantic violations through static bug detection.*

4.3.5 Manifestations

Timing of Violation Understanding when semantics are violated can shed light on how to detect the violation.

As Figure 4.6 shows, some semantics only exist during the execution of its associated operation (at return point), *e.g.*, read operation should return the latest data. We call them *short-lived* semantics. In comparison, some semantics exist even after its associated operation finishes, *e.g.*, the specified file in create operation should be persisted and continue to be available after create returns. They often only cease to apply after some other event, *e.g.*, until a delete operation on the same file is executed. We call them *long-lived* semantics.

Interestingly, we find that 67% of the cases violate long-lived semantics. This

is partly because these semantics have a larger “vulnerability” window compared to short-lived semantics: a violation can occur anytime in its lifespan. ZooKeeper ephemeral znode and watches are such examples. Essentially, the system must **maintain** the promise for a long time.

We categorize the violation timing into four scenarios: at the end of short-lived semantics (①), *e.g.*, wrong response, at the start (②) or in the middle (③) or near the end (④) for long-lived semantics. An example for ② is in HDFS-12217 the snapshot operation did not capture all open files, which violates the long-lived snapshot semantics since the beginning. An example for ③ is HDFS-9083: at the block creation time, the block placement policy is honored; but after some node failures, all replicas of the block reside on the same rack. ④ happens when the semantics should cease to apply but did not, *e.g.*, ephemeral nodes should be removed when clients timeout. Figure 4.6 shows the distributions of the four scenarios.

Finding 9: *Near two thirds of the studied cases violate some long-lived semantics. In 40% of the cases, the semantics are initially honored but are violated in the middle.*

Implications: *It is crucial to continuously monitor semantic guarantees, even after the initial semantic check passes.*

Failure Triggering Conditions We further examine what triggers the semantic failures. Figure 4.7 shows the result.

Finding 10: *More than half of the studied failures are triggered by specific requests, while 39% of the failures require particular timing to trigger. Semantic failures often (41%) only manifest themselves under multiple types of conditions.*

HDFS-14514 is an example of semantic failure that requires multiple *types* of

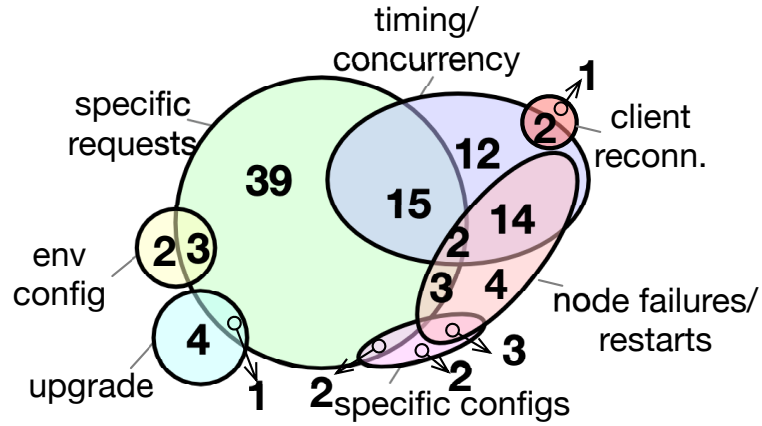


Figure 4.7: Distribution (# of cases) of the failure triggering conditions. Some combinations are omitted in the diagram for readability.

triggering conditions. The semantic violation (read out file size from snapshot is incorrect) can be only triggered when 1) `snapshot.capture.openfiles` is set; 2) create empty directory and encryption zone; 3) a client keeps a file open for write under the empty directory; 4) append several times; 5) perform a maintenance operation, snapshot.

We also find that in 23% of the cases, the triggering condition is certain system maintenance operation, such as compaction, cluster upgrade, node decommissioning. Such events do not occur frequently. They trigger semantic violations often because during the maintenance operation, the system execution enters a different mode, which exposes rare bugs.

Implications: *The reliability of semantics is vulnerable to maintenance operations or node events. Operators and the system should check violations during and after such actions.*

4.3.6 Current Practice

4.3.6.1 Testing

Since semantic violations concern functionality correctness, testing is responsible for catching them. The prevalence (Section 4.3.2) of many semantic failures in production seems to suggest a lack of testing. But that is not the case. The systems we study have extensive test cases—a median of 1309 test files. In addition, in 73% of the studied cases, the system has at least one test case covering the violated semantics.

Then *why the studied failures are not exposed during testing?* The earlier Finding 10 provides some clues. In many cases, even though there are related test cases, they lack some operations or arguments key to trigger the production failure. Even when the test cases have the proper operations and arguments, they only exercise the system under one timing, one configuration or normal scenarios, while the bugs are only triggered with unique timing, configuration, or node failures.

Are the failure triggering conditions so special that it is impossible for developers to foresee? Interestingly, we find that in many cases, similar triggering scenarios do exist in the test suite but they are not used in testing the violated feature.

Finding 11: *Semantic violations occur not simply due to a lack of testing. The violated semantics are usually (73%) covered by some existing test. In more than half of the studied failures, similar triggering conditions exist in the test suite.*

A fundamental gap is that developers tend to write tests driven by examples or fixes for a specific bug. Such tests are not expressive enough to preserve the underlying semantics and prevent regression. Consequently, developers spend repeated efforts to add tests. In HDFS-14514, the server reads snapshot file with incorrect length from encrypted zones. This exact semantics is already checked in an existing test

case. If that test “copies” one line of test configuration `dfsAdmin.createEncryptionZone(...)` from other tests, the new bug will be triggered and exposed.

Implications: *Coverage of semantics alone is insufficient. Developers should introduce variances in existing test cases. It is also useful to “copy” triggering conditions across tests. More fundamentally, developers should write more general tests for the semantic properties rather than specific examples.*

4.3.6.2 Assertions

Assertions are a common method for catching logic bugs, which are major contributors to semantic failures (Section 4.3.4). They are typically only used in development and are turned off by default in release build for performance and stability.

Some of our studied systems use assertions in production: MongoDB has added many invariant checks since 2014 [173]. Interestingly, as Section 4.3.2 shows, MongoDB has the lowest ratio of semantic failures compared to other systems. While this practice may cause instability, *e.g.*, some users got infrequent crashes due to invariant check failures after upgrading to new versions [174], developers still prefer to fix the underlying bugs rather than turning off assertions completely.

We observe two gaps in the current practice. First, most existing assertions are pre-condition checks on the sanity of function arguments. They are too low-level to catch semantic violations, which require checking system functionalities and usually the operation history (*e.g.*, in checking consistency violations [155]). Second, existing assertions are usually only activated once during an operation, *e.g.*, the entry of a function. But many semantics are long-lived (Section 4.3.5), which require continuous validation until the lifespan of semantics ends.

Finding 12: *Although in 51% of the failures the buggy functions have some sanity*

checks, few (9%) cases can be potentially detected by adding proper sanity checks.

Implications: *Enabling assertions helps reduce silent semantic violations. However, developers should add more semantic-level invariant checks besides sanity checks.*

4.3.6.3 Observability

Since our studied failures are silent violations, *how do users notice these subtle failures then?* Understanding this question can reveal insights to improve the observability of semantic failures. We carefully examine the discussion threads in each ticket. In 34 cases, users mentioned their experience clearly.

For all of these cases, users discovered the issues through noticing something suspicious in some “side channels”. We categorize them into two types: (i) benign errors in other requests (32%); (ii) anomalies in logs, files, or performance of other tasks (68%). In HBASE-11654, users find out the violations by noticing splitting directories in `/hbase/WALs/`, which is “very strange” because “*those logs should have been replayed and deleted*”. In KAFKA-9137, users observe the failure by seeing an increase in eviction rate in the logs. In CASSANDRA-6527, users found tombstones appeared even though they never used delete for a column family.

It might seem that we can rely on users to manually detect system semantic failures. Note that there is a survival bias: our studied cases by definition are identified, but in practice silent semantic violations can be easily missed because (i) users do not monitor the systems 24×7; (ii) when they check, they may not inspect the proper signals. When users notice the failures, the damage may be already done. In CASSANDRA-6527, users commented: “*Fortunately, we have noticed that quickly and canceled the migration. However, we were quite lucky.*”

How to make semantic failures more observable? First, if a system API has no

interaction with others, it is hard to judge its correctness based on a single piece of information. In practice users often use multiple related APIs to cross-compare results. In HBASE-15236, users observe the violations because Get and Scan return different sizes for the same bulkloaded hfiles. Second, current systems often do not expose enough information about their internal states, thus users have to *ad-hocly* infer whether a promise is obeyed or not. Existing error messages (*e.g.*, a legitimate exception for another request) only focus on the current request, which is hard to link to the semantic violation in past correlated requests.

Finding 13: *Semantic violations are currently observed from “side channels”: 32% from errors in other requests, 68% from anomalies in logs, files or performance of other tasks.*

Implications: *Designs of overlapping APIs improve observability of semantic violations. Systems should provide more admin APIs for convenient query of their internal states. Error messages should provide hints about past correlated requests.*

4.4 Oathkeeper: Checking Semantic Violations

Guided by our study, we build a tool *Oathkeeper* to check semantic violations for large-scale distributed systems.

4.4.1 Design Overview and Workflow

Oathkeeper takes a runtime approach to check semantic violations in production. This choice is motivated by our findings that semantic failures have diverse root causes (Finding 8) and often difficult to expose in testing due to complex triggering conditions (Finding 10).

Central to a runtime verification approach is what invariants to use. Existing solutions rely on users to write distributed assertions to check the correctness of distributed protocols [148, 147] or network functions [216]. In those scenarios, the semantics to check are limited and well-defined. But in our cases, the systems have abundant (Table 4.1) and loosely-defined semantics. Even for semantics that can be described in simple expressions informally, mapping them to the concrete checkable invariants in the complex systems code is hard. These factors make manual construction a daunting task.

Insight and Key Idea. The insight behind Oathkeeper is based on our finding that the majority of the studied failures violate old semantics (Finding 5) despite the decent coverage of testing (Finding 10). When a semantic failure occurs, developers usually add regression tests. But these tests only check if the specific bug is fixed in a specific setup, while the same semantics can be violated repeatedly in other scenarios.

Based on this insight, Oathkeeper leverages the existing regression tests developers write for past semantic failures and automatically extracts the essence—the violated semantic rules. Oathkeeper then enforces these rules at runtime to detect future semantic violations, which may be caused by different bugs under different conditions.

Input and Output. To apply Oathkeeper to a new system, users supply a system-wide configuration and a list of past semantic failure metadata. The former provides basic information about the system such as the compilation command and test directory, and optionally the classes to include for analysis. The latter metadata is provided in the form of git commit id (for version switching) and regression test name.

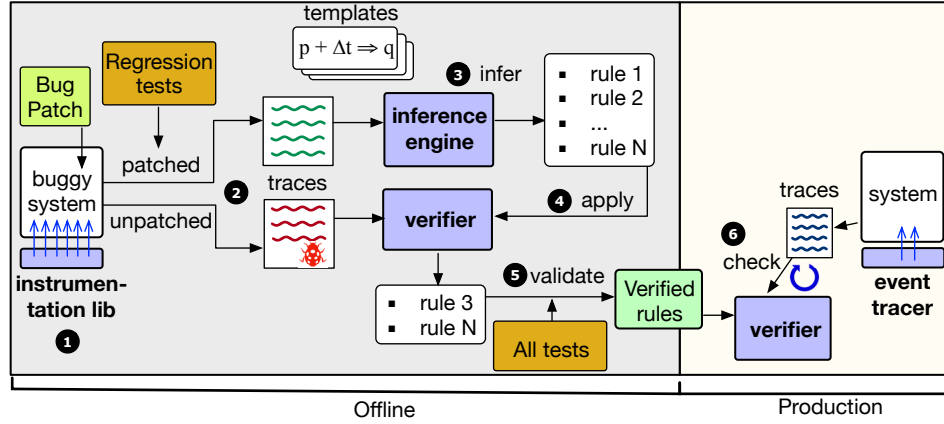


Figure 4.8: Workflow of Oathkeeper.

Oathkeeper outputs the likely semantic rules (Section 4.4.3). Prior runtime verification tools focus on invariants expressed as predicates among key state variables in a system such as `lock_id` and `lock_mode`. This representation alone can be insufficient or complex to express the semantics of large distributed systems. Instead, Oathkeeper focuses on rules that describe relations among semantics-related events, particularly operation invocations and state updates. Such an *event relation* rule is expressive to capture various semantics.

Workflow. Figure 4.8 shows the tool’s workflow. Oathkeeper operates in two stages. In the offline stage, Oathkeeper instruments the target systems to record major events (❶). It then exercises the system twice with the regression tests: once using the patched version and the second time using the buggy version. This will generate two sets of traces (❷). The *inference engine* infers likely semantic rules from the traces of the patched version (❸). The *verifier* applies the inferred semantic rules against the traces of the buggy version and output rules that are violated in the buggy traces (❹). We assume these violated rules are potentially related to the semantic failure. Further optimizations are applied to remove noises and redundancies (❺). In the online stage, Oathkeeper only performs minimal instrumentation that is relevant

to these final semantic rules from the offline stage. The event tracer ingests traces from the system in real time. The Oathkeeper verifier continuously checks the traces against the deployed semantic rules and reports violations (⑥).

4.4.2 Instrumentation and Trace Generation

For both inferring semantic rules and runtime verification, we need to first instrument the system to obtain execution traces. The Oathkeeper traces use a uniform *event* schema that captures operation-related events and state-related events.

Oathkeeper designs a *load-time instrumentation library* that performs bytecode manipulation when a target system is loaded. This way of instrumentation is convenient (without re-compiling and re-packaging the system) and transparent.

To record operation events, the library adds hooks at the beginning, return and exception point of a method. To record state events, Oathkeeper takes a patch plus base approach. It analyzes the given semantic failure patch and automatically includes the list of classes involved in the patch file. Users can optionally specify names of some important system classes, such as `SessionTrackerImpl`. With the combined list of classes, Oathkeeper performs simple analysis at the loading phase of these classes to retrieve their member variables of primitive or collections types, and treat them as the state variables. It then identifies instructions that *update* these variables and insert a hook to emit a state update event with the relevant context (variable name, location, *etc.*).

For each given test, Oathkeeper switches the target system to the patched version. The tool executes the test with the instrumented system and generates the trace of events. Then Oathkeeper reverts the target system to the buggy version (snapshot prior to the patch commit id). Since the buggy version does not contain the test,

Template	Example
$p \Rightarrow q$	decommission a datanode should trigger reconstruction
$s \uparrow \Rightarrow p$	when datanode changes, associated watcher notifies clients
$s \uparrow \Rightarrow k \uparrow$	after session disconnection, ephemeral node is removed
$(s = c) \oplus q$	read-only server should not provide write access
$p + \Delta t \Rightarrow q$	inserted data should expire after the TTL is reached.
$s \uparrow \rightarrow q$	cf schema should be altered before alter command returns
$p \Rightarrow \odot(s \uparrow, k \uparrow)$	after snapshot renaming, either new snapshot creation and old snapshot deletion both occur or none of them occur

Table 4.3: Some templates integrated in Oathkeeper.

Oathkeeper copies the regression test from the patched version and executes it to get the buggy trace. If the test cannot directly run on the buggy version due to interface changes (*e.g.*, a function used in the test is not public in the buggy version), the tool supports user-provided patches to fix the compatibility issue.

The trace is stored in a JSON file for ease of deserialization. An example trace entry is `{"type": "OpTriggerEvent", "data":{"opName": "zookeeper.FileSnap.deserialize", "time": 1654026992, ...}}`. The trace scale is usually moderate, because it is generated from tests. For example, with ZooKeeper, even under the full instrumentation mode (instrumenting all classes), most end-to-end tests generate less than 10,000 events. A common scale is several thousands. We see large traces in only 5/273 tests that produce over 500,000 events. Under the diff mode (only instrument the classes affected by the patch), the trace typically has hundreds of events.

4.4.3 Template-Driven Inference

A key challenge in the semantic rule inference step of Oathkeeper is to integrate domain knowledge without requiring significant manual effort, while also having reasonable accuracy and efficiency. We take a template-driven approach to address this challenge. We first summarize general semantic rule patterns, such as happens-before relationship, atomicity, periodicity. For each pattern, we define one or more parameterized templates, such as a state change event for s must happen before the completion event of operation p . Oathkeeper currently supports 18 templates. Table 4.3 shows several examples. p, q are operations, s and k are states, t is time, c is constant. $\neg p$ means p can not occur. \uparrow means state changes. $p + \Delta t$ means time t after p occurs. Our technical report [152] shows the full list.

The inference engine implements an inference algorithm for each template. The algorithm checks if there are matches in a given trace and derives concrete values to each template parameter if so. We call each match a *context* for the template, which is a potential invariant. For one template (e.g., $p \Rightarrow q$), a trace can have multiple contexts (e.g., $a1 \Rightarrow a2$ and $a1 \Rightarrow a3$).

The templates allow encoding domain-specific semantics without significant specification effort. They also restrict the search space so the inference engine only analyzes trace events that match the template structure and parameter types. While these templates may not represent the exact or full semantics like a high-level specification does, they can capture the essential ingredients for making the semantics hold.

The inference engine takes the trace obtained from running the regression tests against the patched system. Each template class implements an `infer` function that returns a list of rules from the trace. Most templates follow three phases in the `infer`

```

public abstract class InferScanner {
    //init state variables
    abstract void prescan(Set<Event> eventSet);
    //always need to go through the whole traces
    abstract void scan(Event event);
    //check states after scan, and generate invariants
    abstract List<Invariant> postscan();
}
public abstract class VerifyScanner {
    //init state variables
    abstract void prescan();
    //return true if continues to scan, otherwise break
    abstract boolean scan(Event event, Context context);
    //check states after scan, and judge
    abstract InvState postscan();
}

```

Listing 1: Inference and validation interfaces for each template.

function: *pre-scan*, *scan*, and *post-scan* (interfaces defined in Listing 1). The pre-scan step typically builds an index of the unique event set in the trace. The uniqueness is determined by a custom function we define for different types of events. For example, operation invocation events are unique based on the signatures of invoked functions. The scan step iterates through each event in the trace and updates bookkeeping data structures such as an event occurrence map. The post-scan step generates invariants based on the bookkeeping data structures. Templates that do not follow this pattern can customize the procedures. For example, the `AfterOpAtomicStateUpdateTemplate` iterates forward once and scans backwards once; the `StateEqualsDenyOpTemplate` scans the trace for each state type in the test.

The core inference algorithm for each template, while different, is relatively straightforward. It essentially involves identifying events in the trace that match the type of a template’s parameter, enumerating hypotheses (candidates) from the contexts, and validating the hypotheses against the trace. Since the trace size is moderate, we can afford enumerations.

Algorithm 1: Generic inference and validation workflow.

Input: L : a trace (list of events)

Output: a list of inferred invariants (one inv. is a template w/ context)

Func Infer(L):

```
/* get unique events in the trace (we define equality individually for different
   types of events) */
unique_events  $\leftarrow$  Set( $L$ )
prescan(unique_events)
foreach event  $\in L$  do scan(event)
return postscan()
```

Input: L : a trace (list of events), $context$: parameters in templates, *e.g.*, if an invariant is $a1 \Rightarrow a2$, context is $a1$ and $a2$

Output: the checking result of invariant (pass, fail or inactive)

Func Verify($L, context$):

```
foreach event  $\in L$  do
    if scan(event, context) then break
end
return postscan()
```

Func Main($L_{patched}, L_{buggy}$):

```
inv_list  $\leftarrow \emptyset$ 
foreach inv  $\in$  Infer( $L_{patched}$ ) do
    if Verify( $L_{buggy}, inv.context$ ) == InvState.FAIL then inv_list.add(inv)
end
return inv_list
```

Example. We describe the inference of a representative template $p \Rightarrow q$, which represents that every invocation of operation p implies a subsequent operation invocation of q . For example, `createSession` should usually imply `closeSession`. The steps are listed in Algorithms 1 and 2.

We use Figure 4.9 (a) to show the process of inferring rules of template $p \Rightarrow q$ from a patched trace $[e1, e2, e3, e1, e2]$. The algorithm assumes all pairs $\langle e_i, e_j \rangle$ in the unique event set are candidate contexts to the template, in which e_i and e_j are of `OpTriggerEvent` type and the uniqueness is based on the operation name. Then it attempts to find counterexamples to invalidate wrong rules. The inference algorithm of this template uses a simple counting approach that runs in three steps. The *pre-scan* step constructs a nested map $\{event: \{event: int\}\}$ to record the occurrences

(a) inference		pre-scan		<e1,e2>	<e2,e1>	<e1,e3>	<e3,e1>	<e2,e3>	<e3,e2>
		scan	e1	1	0	1	0	0	0
			e2	0	1	1	0	1	0
			e3	0	1	0	1	0	1
			e1	1	0	1	0	0	1
			e2	0	1	1	0	1	0
		post-scan		e1=>e2			e3=>e1		e3=>e2
(b) validation		pre-scan		<e1,e2>		<e3,e1>		<e3,e2>	
		scan	e1	1		0		0	
			e2	0		0		0	
			e3	0		1		1	
			e1	1		0		1	
			e1	1		0		1	
		post-scan		e1=>e2		/		e3=>e2	

Figure 4.9: Inference and validation algorithm example.

for the event pairs. For each event pair, the counter is initially zero. Then the *scan* step iterates through each event e in the trace in order. If e is e_i , *i.e.*, a key in the nested map, we increment the counters for all entries with $\langle e_i, * \rangle$ keys; if e is e_j , we decrement the counters for entries that have $\langle *, e_j \rangle$ keys *and* have positive counters. In the *post-scan* step, we check the final state of counters. If the final counter does not reach zero, there is an orphan e_i that does not have subsequent e_j . We get $e1 \Rightarrow e2$, $e3 \Rightarrow e1$ and $e3 \Rightarrow e2$ at the end. Rules like $e1 \Rightarrow e3$ are removed because no subsequent $e3$ occurs after the second $e1$.

4.4.4 Rule Validation

After step ③, the inference engine could infer many likely semantic rules. Oathkeeper then applies these rules against the buggy traces (④) and sees which rules are violated. Similarly to inference, each template class needs to implement a *verify* function. The *verify* function also usually consists of three phases: *pre-scan*, *scan*, and *post-scan*. The *pre-scan* step initializes auxiliary data structures specific to the template. The *scan* step goes over the events in the trace and updates the data structures. In some template, the *scan* step does not need to iterate through all events in the trace if a contradictory example is already found. The *post-scan* step checks the data structures

and returns the result, which could be PASS (rule is activated and no contradiction is found), INACTIVE (the antecedent of the rule does not occur, *e.g.*, $p \Rightarrow q$ is inactive in a trace without occurrences of p), or FAIL (at least one contradiction is found). We only preserve rules that pass in the patched trace and fail in the buggy trace.

Example. Algorithms 1 and 2 show the steps to verify template $p \Rightarrow q$. We use Figure 4.9 (b) to show the process of validating inferred rules from (a) on a buggy trace $[e1, e2, e3, e1]$. There are three rules to verify: $e1 \Rightarrow e2$, $e3 \Rightarrow e1$, $e3 \Rightarrow e2$. In the *pre-scan* step, we first initialize a counter for each inferred rule. The *scan* step then updates the counter: for rule $e_i \Rightarrow e_j$, if a processed event e matches e_i , we increment the counter; if e matches e_j , we decrement the counter if it is positive. All three rules are active as both $e1$ and $e3$ appear in the trace. The *post-scan* step marks rules with non-zero counters as FAIL: $e1 \Rightarrow e2$ and $e3 \Rightarrow e2$.

However, there could still be a significant number of rules due to noises like unfinished tests (*e.g.*, an assertion failed in the middle of the test), new type events (new methods introduced), coincidence, and methods that are used for testing only. To reduce these noises, the verifier validates (5) the candidate rules against traces obtained from all test cases, under the patched version, and *discards* rules that do not hold in all traces. In addition, we filter uninteresting rules about the system start-up or shut-down methods or thread run methods. This is achieved by inserting special marker events at the start and end of test method, and only running the inference algorithms on trace region within the markers.

4.4.5 Runtime Checking

Oathkeeper deploys the refined semantic rules with the target system in production, along with the verifier and event tracer. Oathkeeper performs load-time instrumentation to the production system in a wrapper class of the entry points. Different from the offline stage, the instrumentation is selective to only the deployed rules and is thus lightweight. The event tracer stores in-memory traces from the target systems.

The runtime verifier schedules periodical tasks that validate the current trace against *each* of the deployed semantic rules. It reuses the same checking logic defined in the function `verify` of the template. When the engine finds a semantic rule reported as `FAIL`, it records the counterexamples in the traces for debugging. It also schedules a second check on this violated rule again shortly to tolerate transient violations or inconsistencies in the trace. For high availability, Oathkeeper generates alerts in the log upon detection of potential semantic failures and does not attempt to crash the system.

4.4.6 Optimizations

The validation step can be time-consuming. With N (often thousands) candidate rules and M (often hundreds) test cases, we need to get M traces and check $N \times M$ times. To reduce the validation time, we introduce a survivor optimization. After a test finishes, we validate the rules, if some rule is already “killed” (invalidated) by this test’s trace, it will not be carried over to the remaining tests. Therefore, only the survived rules will be validated to the end. Another optimization is to run more closely related tests first. The rationale is that some test takes a long time to run but is irrelevant to a given rule (thus the test’s trace will not disprove the rule). By

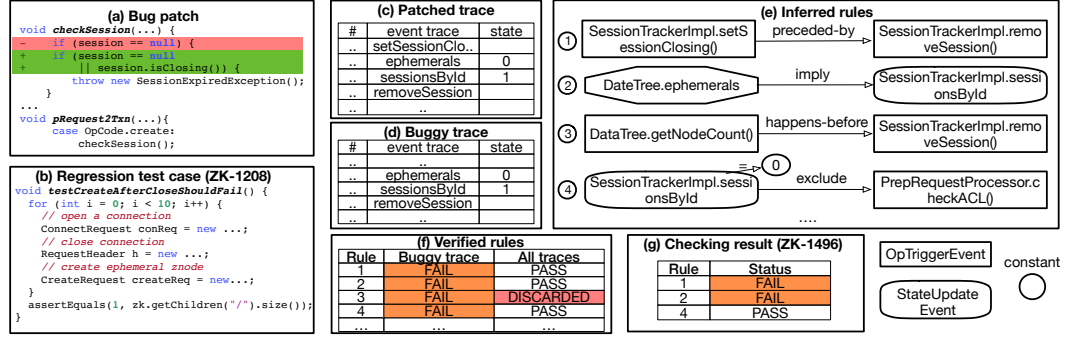


Figure 4.10: Example: Oathkeeper workflow of using ZK-1208 to detect ZK-1496.

prioritization, we can potentially invalidate false rules faster.

We also add several optimizations to reduce the runtime overhead. First, the event tracer only preserves the most recent events within a time window, since always checking full traces from the start is wasteful. The time window is configured larger than checking frequency to avoid missing checking events. The events involved in time-related semantic rules are excluded as their expiration time is based on their parameters. Second, to achieve both high concurrency and low memory pressure, we decouple the checking from the event emission with a ring buffer design inspired by high-performance message queues [149]. Third, to avoid massive new object creation frequently triggering garbage collection, we reuse expired event objects in the ring buffers. Oathkeeper also pre-allocates buffers for each type of events at the instrumentation phase to prevent buffer initialization blocking.

4.4.7 Implementation

We implement Oathkeeper in Java (JDK 8). Its instrumentation library is built based on Javassist for class bytecode manipulation. Its test engine leverages JUnit to manage and execute test cases. The tool also includes a workflow script such as checking out patched and buggy versions and checking a semantic rule against given traces.

4.4.8 Limitations

Our approach makes several assumptions: 1) semantics should be expressible with simple relations of events; 2) the system has a number of test cases with good quality; 3) the failure patch should not involve significant redesign or interface interfaces. If some assumption does not hold, Oathkeeper may fail to deduce good semantic rules.

4.5 Evaluation

We have integrated Oathkeeper with ZooKeeper, HDFS and Kafka. We evaluate (1) whether Oathkeeper can leverage past semantic failures to check new violations; (2) what runtime overhead it incurs to the target system. The experiments are done in servers with 20-core 2.2 GHz CPUs, 64 GB memory, running Ubuntu 18.04. The Oathkeeper check engine is configured to schedule and check rule violations every second.

4.5.1 Generation Overview

Oathkeeper requires old semantic failures and their associated regression tests as input to extract semantic rules. We select old semantic failures and their regression tests to reproduce (8 for ZooKeeper, 10 for HDFS and 8 for Kafka). These tests cover major functionalities of the three systems. We add a switch in the system code to easily enable and disable the patch for the semantic failure bugs. We then apply Oathkeeper to the source code to add instrumentation points, run the regression tests with the patch switch turned on and off, and execute other steps in Oathkeeper (Section 4.4.1). For each case, Oathkeeper infers many raw semantic rules. After the validation and optimization step, the rule set is significantly reduced. In total,

Oathkeeper extracted 285 rules for ZooKeeper, 1,209 rules for HDFS, and 150 rules for Kafka.

4.5.2 Checking Newer Violations

We evaluate whether the inferred rules are useful to catch new semantic failures. Given Oathkeeper’s approach, it is likely less effective with unseen semantics. We reproduce 7 newer (9–34 months later) failures (Table 4.4) that violate related semantics in the old cases, but with different root causes. With the inferred rules, Oathkeeper detects violations for 6 of them. These newer violations are known bugs by the time we conducted this experiment. However, their root causes and triggering conditions are completely different from the failures used to extract semantic rules. Oathkeeper detects these newer violations with only knowledge from the old failures, which demonstrates the tool’s detection capability.

We show one example in Figure 4.10. ZK-1496 is not in our study dataset, but its symptom is similar to a studied failure ZK-1208 that was reported 9 months ago prior to ZK-1496 in an older release. Users found that the ephemeral znodes were not deleted long after the client exited. The root cause is a race condition bug that while the session tracker is removing the expired session, another thread is processing an ephemeral node creation request. In ZK-1208, developers added a fix to mark sessions as closing to prevent ephemeral node creation on expiring sessions, and introduced a regression test. Oathkeeper executes the regression test on ZooKeeper twice with patch enabled and disable, and generates two traces (c) and (d). Then Oathkeeper infers rules (e) from the patched traces. Not all inferred rules are useful. Oathkeeper only preserves rules that fail in buggy traces and pass all tests (f). Rules such as ③ are filtered when being validated on all tests. Finally, two verified rules ①

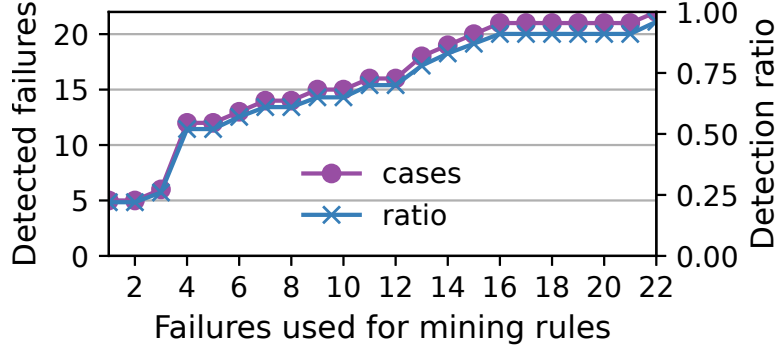


Figure 4.11: Detection of 22 semantic failures in ZooKeeper (sorted by the bug ticket time in ascending order) when applying Oathkeeper on a sliding subset of the failures for inferring semantic rules.

and ② detect the violations (g).

Oathkeeper fails to detect ZK-1667: client A sets a watch on /d and then disconnects, client B deletes /d and recreates it; when client A reconnects, it receives a NodeCreated event instead of NodeDataChanged event. The violated semantics fits into one of our templates. However, due to the quality of the old watch test in our pool, Oathkeeper infers other rules.

The average detection time is 0.91 seconds. This result does not contradict with the long-lived semantics finding in Section 4.3.5. In the experiments, we trigger the conditions to reproduce the failure soon and measure the detection time from the start time of the violation.

We compare Oathkeeper with a state-of-the-art invariant checking tool, Dinv [80]. Dinv is designed for checking distributed protocols. Its core invariant inference component is based on Daikon [61] that mines variable-level relationship. We instrument the state variables in the two systems and apply Daikon to traces from the system test cases. The inferred invariants only detect 1 case (ZK-1496) and are highly noisy.

We conduct an additional “cross-validation” experiment. Specifically, we collect a larger pool of 22 semantic failures in ZooKeeper. The failures are sorted from

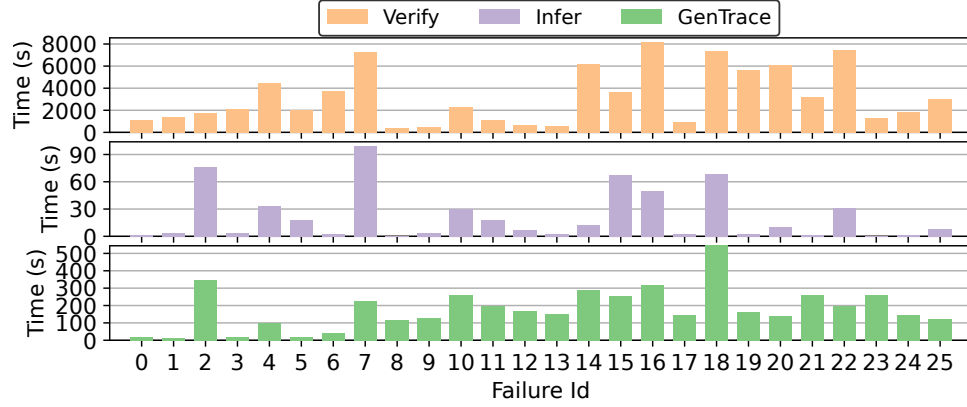


Figure 4.12: Time to generate trace, infer rules, and verify rules against test suite.

older to newer. We feed each failure to Oathkeeper and measure how many of the 22 failures can be detected. For 16 cases, the rules inferred from one case only detect that case. It does not imply, though, these rules are useless. They might help detect failures outside the pool. Interestingly, for the remaining 6 cases, their inferred rules detect a median of 5 failures. For example, rules from ZK-2355 can detect 6 other failures besides itself. Figure 4.11 plots the aggregate detection result.

4.5.3 Performance

Figure 4.12 shows the performance of running Oathkeeper for the 26 old cases. Our template-based inference is fast. The median time to finish inference is 6.5 s. The median trace generation time is 153.5 s. The most time-consuming part is verifying the inferred rules against the system test suite, because running the full test for the three systems alone takes a long time. The end-to-end validation time is 2196 s (median). After discounting the original test execution time, the median validation time is 301 s. The survivor optimization we introduce (Section 4.4.4) helps. In one time-consuming case, it reduces the end-to-end validation time from 8104 s to 5024 s.

4.5.4 Runtime Overhead

We measure the overhead Oathkeeper introduces to the systems at runtime. The main source of overhead comes from the added instrumentation to emit traces; the rule checking does not impact the system much because it is done asynchronously. Oathkeeper only adds instrumentation relevant to the deployed rules to minimize the overhead. Naturally, more rules lead to higher overhead. We evaluate the overhead as a function of the percentage of enabled rules. For ZooKeeper, we run the workload of 15 clients sending 15,000 requests (40% reads, 60% creates and writes). For HDFS, we run the built-in benchmark NNBenchmarkWithoutMR which creates and writes 100 files, each file has 160 blocks and each block is 1MB. For Kafka, we run the workload of producing 1 million 16KB messages. Table 4.5 shows the result. With all rules enabled, the average system throughput overhead is 1.27%.

Our initial event tracer used an array list with synchronization, which resulted in a 31% overhead under heavy workloads. We later implemented a more complex non-blocking queue, but the overhead is still large. After investigation, we found the overhead mainly comes from memory and GC instead of synchronization, which motivated our ring buffer design (Section 4.4.6) that significantly reduced the overhead.

4.5.5 Rule Activation and False Positive

We deploy the inferred rules to a cluster of ZooKeeper, HDFS, and Kafka instances. We run a set of workloads against the instances. We first measure the rule activation ratio during the experiment. A rule is activated if the check engine finds the antecedent of the rule has occurred. For ZooKeeper, 11% of the rules are activated. The remaining rules are not activated due to the lack of workloads, faulty conditions,

etc., to trigger the antecedent events. For HDFS and Kafka, the activation ratio is 66% and 48%. We then measure the false positive ratio among the activated rules. The result is 4% for ZooKeeper, 9% for HDFS, and 12% for Kafka. This result benefits from the validation steps described in Section 4.4.4: Oathkeeper eliminates falsely inferred rules by validating the rules against both the buggy trace and the traces from all test cases of a target system. Adding profile runs or a dynamic ban mechanism can further remove the false rules.

4.6 Conclusion

Silent semantic violations pose a severe challenge to distributed systems reliability. This chapter sheds light on this under-explored yet important problem by presenting a study on real-world failures in popular distributed systems. It reveals that sadly “a promise is often *not* a promise”. Guided by our study, we design a tool Oathkeeper that automatically extracts semantic rules from past semantic failures, and enforces these rules at runtime to check future violations.

Algorithm 2: Implementation for template $p \Rightarrow q$.

```
Func ImPLYTemplate::InferScanner::prescan( $S$ ):  
  foreach  $event \in S$  do  $C.put(event, \{\})$   
  foreach  $event \in S$  do  
    foreach  $event2 \in S$  do  
      if  $event \neq event2$  then  
         $C.get(event).put(event2, 0)$   
         $C.get(event2).put(event, 0)$   
      end  
    end  
  end  
Func ImPLYTemplate::InferScanner::scan( $event$ ):  
  foreach  $(k, v) \in C.get(event)$  do  $v \leftarrow v + 1$   
  foreach  $event2 \in C$  do  
    if  $event == event2$  then continue  
     $val \leftarrow C.get(event2).get(event)$   
    if  $val > 0$  then  $C.get(event2).put(event, val - 1)$   
  end  
Func ImPLYTemplate::InferScanner::postscan( $L$ ):  
   $lst \leftarrow []$   
  foreach  $(k, v) \in C$  do  
    foreach  $(k2, v2) \in v$  do  
      /* add potential invariants when counter is 0 */  
      if  $v2 == 0$  then  $lst.add(genImPLYInv(k, k2))$   
    end  
  end  
  return  $lst$   


---

Func ImPLYTemplate::VerifyScanner::prescan():  
   $ifHold \leftarrow true$   
   $ifActivated \leftarrow false$   
   $counter \leftarrow 0$   
Func ImPLYTemplate::VerifyScanner::scan( $event, context$ ):  
  if  $event == context.left$  then  
     $counter \leftarrow counter + 1$   
     $ifActivated \leftarrow true$   
  else if  $event == context.right \ \&\& \ counter > 0$  then  
     $counter \leftarrow counter - 1$   
  end  
  return  $true$   
Func ImPLYTemplate::VerifyScanner::postscan( $L$ ):  
  if  $counter \neq 0$  then  $ifHold \leftarrow false$   
  if  $!ifHold$  then return  $InvState.FAIL$   
  if  $ifActivated$  then return  $InvState.PASS$   
  else return  $InvState.INACTIVE$ 
```

JIRA Id	Violated Semantics
ZK-1496	ephemeral node should be deleted after session expired
ZK-1667	watcher should return correct event when client reconnected
ZK-3546	container node should be deleted after children all removed
HDFS-14699	failed block need to be reconstructed
HDFS-14317	edit log rolling should be activated periodically
HDFS-14633	file rename should respect storageType quota
KAFKA-12426	partition topic ID should be persisted into metadata file

Table 4.4: Evaluated newer semantic failures.

	Base	25%	50%	75%	100%
ZooKeeper	418.27	417.63	416.71	416.55	416.1
HDFS	174.55	174.56	172.10	172.10	172.06
Kafka	30,759.49	30,546.00	30,377.50	30,246.04	30,183.15

Table 4.5: System throughput (op/s) with varying percentages of semantic rules enabled.

Chapter 5

Memory Leaks at Cloud-scale

5.1 Introduction

Memory leak is a prevalent issue in software, from applications [67] to OS kernels and device drivers [209]. At Microsoft Azure, its infrastructure contains many complex software components running on a massive number of machines with various workloads. Unsurprisingly, these components encounter memory leak issues from time to time. When a process leaks memory, the direct consequence is performance degradation and crash. Worse still, its impact often affects other components running on the same machine, such as causing excessive paging, innocent processes being killed, and node reboots.

Memory leak is notoriously difficult to deal with, especially in a production cloud infrastructure setting. The issues are usually only triggered by rare conditions and occur slowly, thus they easily escape testing and failure detectors [121]. After leak symptoms are detected, it is time-consuming and sometimes impossible to reproduce them offline. Unlike other failures like crashes that have clear points to start diagnosis, developers are often clueless in finding the leak's root cause.

Extensive solutions have been proposed to detect memory leak bugs. One approach uses static analysis techniques [212, 113, 87, 177, 62] to analyze the software source code and deduce potential leaks. The second approach detects memory leaks dynamically by instrumenting a program and tracking the object references at runtime [91, 184, 213, 139, 125].

While helpful, these solutions are insufficient to address the memory leak challenges in Azure. Static approach is limited by the well-known accuracy and scalability issues with static analyses. It also only focuses on leaks in which allocated objects are unreachable [128]. If memory objects are reachable but never accessed again, it still incurs the consequences of leaks. Such leaks are hard to detect statically. Moreover, memory leaks in cloud infrastructure can be caused by cross-component contract violations, which require too much domain knowledge to recognize statically.

Dynamic approaches better fit Azure’s requirements. However, while the existing dynamic detection solutions are generally more accurate, they are intrusive and require extensive instrumentations that are cumbersome to apply to complex components [91, 125]. They also incur high runtime overhead that is prohibitive for deployment in production [28].

In this chapter, we present RESIN, an end-to-end service designed to holistically address memory leaks in large cloud infrastructure, from detection to diagnosis and mitigation. RESIN is highly scalable—it analyzes all the host software components, including kernels, drivers, and system processes, on millions of nodes in Azure. RESIN has low overhead while running in production environment. At the same time, RESIN provides good accuracy and helps developers pinpoint the root causes of memory leak issues.

Two key insights motivate the design of RESIN and enable it to achieve the above properties. First, the conundrum of existing solutions is in part because they mix detecting a leak and pinpointing the leak bug in one step, so they have to make trade-offs among accuracy, scalability, and overhead. In our experience, we should decompose the detection and pinpointing into multi-level stages to catch memory leaks at production scale. Second, taking a centralized service approach that leverages low-level system mechanism is essential to support many components transparently in a non-intrusive way. It also enables gathering valuable information from many nodes in the cloud to address the accuracy challenges.

Based on these insights, RESIN performs non-intrusive, low-overhead leak detection first. When a process is suspected of experiencing leaks, RESIN triggers a live heap-snapshot mechanism to capture sufficient evidence and runs diagnoses. RESIN leverages kernel-level monitors and profilers as its building blocks, so it directly supports all the running processes without cumbersome integration. Furthermore, RESIN builds a centralized service that analyzes processes across all hosts in Azure fleet together to capture complex leaks.

A key challenge for dynamic leak detection is the highly noisy nature of memory usage in modern software affected by the workload characteristics. Using simple static thresholds can easily generate many false alarms or false negatives. For instance, in an impactful real-world cloud service outage caused by memory leaks, no alarm was triggered despite the existence of a memory monitoring service [9].

RESIN addresses this challenge by designing a robust *bucketization-based pivot* scheme. It aggregates the memory usages of processes across machines, and groups them into different buckets. Then by performing a pivot analysis on the process name, bucket, and other attributes, RESIN can reliably detect leaks without being prone to

fragile thresholds. Essentially, we focus on analyzing a component’s global memory usage behavior, rather than the microscope of an individual process. The rationale is that a true memory leak comes down to some buggy release. Although the memory usage of an individual process is highly dependent on workloads, the workload effect is likely canceled out when inspecting the usage of the same component running in all machines.

Once a suspicious memory leak is detected, RESIN activates the second stage of taking *live* heap snapshots of the suspected processes, which contain information about the active allocations and their stack traces. This stage is more heavyweight but provides more evidence to help developers confirm and diagnose the issue. Since a leak is often sporadic, RESIN aims to “hit” the leak again and capture useful evidence. It carefully chooses the snapshot time points so that the obtained snapshots have a high chance of localizing the root causes while minimizing the snapshot cost. Besides taking heap snapshots of the suspected leaking process, RESIN performs a *fingerprinting step* that periodically takes heap snapshots of representative processes to build a reference database. This reference database is used in the diagnosis algorithm to further improve the diagnosis accuracy.

Finally, RESIN automatically mitigates a detected leak to minimize its impact on the service availability and performance. The mitigation engine in RESIN leverages the information from the detection and diagnosis engines, and determines the appropriate actions to resolve the leak symptoms while developers investigate the root causes and fixes.

RESIN has been running in production in Azure for more than 3 years. RESIN reported many memory leaks, helped developers diagnose the issues, and automatically mitigated the leaks before their impact becomes visible to customers. Within

the recent year at the timing of writing, the unexpected VM reboots in Azure caused by out of memory are reduced by $41\times$, and the new VM allocation errors due to low memory are reduced by $10\times$. In addition, no severe outages in 2020 and 2021 at Azure were caused by memory leaks.

In summary, the main contributions of this work are:

- A holistic memory leak solution for cloud infrastructure.
- A novel bucketization-based pivot scheme to robustly detect memory leaks with low overhead.
- A live heap snapshot algorithm to effectively capture evidence in production and diagnose memory leaks.
- A lightweight automated leak mitigation design.
- Deployment of RESIN in a production cloud service.

5.2 Background and Motivation

5.2.1 Host Memory Compositions

In IaaS cloud infrastructure, servers are equipped with large memory, a significant portion of which is used by the virtual machines (VMs), while the other portion is used by the host software. The latter includes the hypervisor, host OS kernel, drivers, system processes, and various host agents, *e.g.*, an agent that manages networking of the VMs. In this work, we focus on memory leak issues in host software, not leaks in customer VMs. Unless otherwise specified, the kernel, drivers and processes hereafter refer to those in host software stack.

Leaks in the host software can cause severe performance degradation and even instability of the host OS. They can further impact the running VMs, because memory

between VMs and the host is not strictly partitioned, typically controlled by a soft threshold [208]. They can also cause potential VM start-up failures due to insufficient physical memory available.

The host memory is divided into user-mode memory and kernel memory. The host OS in Azure’s infrastructure distinguishes four states for pages in a process’ virtual memory: *free*, *reserved* (for future use but no physical page is allocated), *committed* (memory has been allocated from physical memory or paging files), and *shared*. For memory leak detection, we only need to consider pages in the committed and shared states. For kernel memory, the kernel creates two types of memory pools: *non-paged* pools and *paged* pools. Virtual memory in the non-paged pool is guaranteed to reside in physical memory as long as the kernel objects are alive, whereas memory in paged pool can be paged out. Memory leaks in the kernel can happen in both types of pools.

5.2.2 Memory Leaks

Memory leak occurs when heap-allocated objects are not freed at appropriate time. It is manifested in two forms: (i) *unreachable* leak, in which an allocated object is no longer reachable from the root objects such as global and stack variables; (ii) *forgotten* leak, in which an allocated object is still reachable but no longer accessed. The first type does not occur in managed languages like Java. For the second type, since the program still keeps references to the leaked object, it cannot be reclaimed even with managed languages [68]. Such leak is challenging to be detected because whether an object will be accessed in the remaining execution is undecidable. Thus, a leak detection solution can only output conservative (correct) answers, *e.g.*, the objects that are definitely dead at a given time point, or approximate answers (which


```

1 // ConfigMonitorThread          5-sec timeout, previously
2 while (cm->running) {           it was set to INFINITE
3     waitStatus = WaitForSingleObject(
4         fileChangeHandle, 5 * 1000;
5     if (waitStatus == WAIT_OBJECT_0) {
6         // object is signaled, config file has changed
7         ::Sleep(200);
8         cm->ReadConfig(); // read the file
9 +     if (!FindNextChangeNotification(fileChangeHandle))
10 +         throw ServerBaseException(
11 +             "Failed to get handle to config directory");
12     }
13 - FindNextChangeNotification(fileChangeHandle);
14 }

```

Figure 5.1: A production memory leak example in Azure from a host process that caused leaks of objects allocated at the kernel side.

may be incorrect) such as inferring based on the object's *staleness* [92].

Memory leaks in cloud software have further complications. For instance, while existing solutions focus on detecting leaks in an individual component, a memory leak in cloud infrastructure often happens because of API contract violations between different components, which is not well addressed. This type of leak is hard to expose in pre-production environment, because software components are often tested separately and integration testing cannot cover all possible interactions. Slow leaks also unlikely get detected due to testing time constraints.

Figure 5.1 shows a real example of such a leak in Azure (this case was successfully caught by RESIN). The process has a thread that monitors the configuration file updates using `WaitForSingleObject` with a 5-second timeout. In each loop iteration, it calls the `FindNextChangeNotification` API (line 13). Each invocation causes the kernel to allocate I/O request kernel objects from the non-paged pool memory. The contract of the `FindNextChangeNotification` is that it must be followed by a call to a wait function, and if the wait function returns for any reason other than the change notification handle being signaled (*e.g.*, timeout), the wait must be re-tried. In this case, although the process calls the wait function, it unconditionally

calls `FindNextChangeNotification` even if the wait returns timeout. Thus, the kernel objects are allocated every 5 seconds without being cleaned up. In this incident, the culprit process' memory usage was not high. The kernel was experiencing memory leak in its non-paged pool, not because of kernel bugs but rather the improper API usage in the process' code. This memory leak was introduced during a bug fix for another issue: previously the process waits for the updates using an `INFINITE` parameter in line 4, but this caused service restart operations to be blocked, so developers changed the wait parameter to a timeout of 5 seconds.

5.2.3 Requirements

There are several challenges and requirements for addressing memory leaks in cloud infrastructure software:

- *Highly scalable.* Cloud system is large in the number of components, codebase size, and deployment scale.
- *Versatile.* Memory leaks in cloud infrastructure manifest themselves in various ways—in processes, kernel, unreachable leaks, forgotten leaks, cross-component leaks, *etc.*
- *Non-intrusive and low-overhead.* Solutions that require intrusive modifications or incur high runtime overhead are hard to be deployed in production.
- *Accurate.* True leaks should be detected. False positives should be minimized, because they would cause developers to waste significant time investigating false issues.
- *Timely.* If the leak detection is too slow, significant damage to customers may already occur.

- *End-To-End.* Only alerting memory leaks is insufficient. Developers also need considerable help in confirming the issue, pinpointing the root cause, and mitigating the leak.

Additional constraints include generality and efforts of integrating a solution. The software components in cloud infrastructure are written in different programming paradigms, and may depend on proprietary libraries. The millions of nodes in Azure also have heterogeneity with different OSes, libraries, and hardware versions. Supporting all of these varieties is challenging. For example, we made an experimental effort of integrating the LeakSanitizer [138], a popular run-time memory leak detector from the LLVM project, into one Azure host component’s codebase. The integration effort was difficult (took one person month) due to complex compilation flags, and library compatibility issues. The MSVC compiler’s full support for LeakSanitizer is still pending [186].

5.3 RESIN: Automating Resolving Memory Leaks at Cloud-scale

5.3.1 Overview

Despite the extensive efforts to address memory leaks in conventional settings, they are insufficient to satisfy the unique requirements for tackling memory leaks in large cloud infrastructure (Section 5.2.3). To address this gap, we propose RESIN. RESIN is a holistic system running in the Azure production infrastructure to detect memory leaks in host software and provide diagnosis support to developers easily pinpointing the leak’s root causes. RESIN further performs automatic leak mitigation to reduce the impact of detected leaks.

Approach A large cloud infrastructure can have hundreds of components owned by different teams. Prior to RESIN, tackling memory leaks in Azure is a team-by-team effort. Some teams started investigating after incident reports about slow or failing VMs, and developers discovered leak bugs in their components during manual investigation. Some teams added telemetry monitors in their testing cluster and used hard-coded thresholds to trigger leak alerts in testing. Similarly, diagnosing leaks in Azure used to rely on developers to manually inspect the leaking nodes and run profiling tools. These individual practices were tedious and costed repeated engineering efforts. They also incurred significant false positives and could not handle cross-component leak issues.

RESIN takes a centralized approach instead. It does not require access to a component's source code, nor extensive instrumentation or re-compilation. RESIN uses a monitoring agent to each host that leverages low-level OS features to collect memory telemetry data. It automatically supports all components including the kernel. The data analysis is offloaded to a remote service, which minimizes the overhead to the hosts. By aggregating data from different hosts, RESIN can run more sophisticated analyses to catch complex leaks.

In addition, RESIN decomposes and tackles the memory leak problem in multi-level stages. It performs lightweight leak detection first and triggers more in-depth inspections on the fly when necessary for confirmation and diagnosis. This divide-and-conquer approach allows RESIN to achieve low overhead, high accuracy, and scalability together.

Workflow Figure 5.2 shows the workflow of RESIN. It starts with low-overhead monitoring (❶) at each host. A remote service analyzes (❷) the collected data across

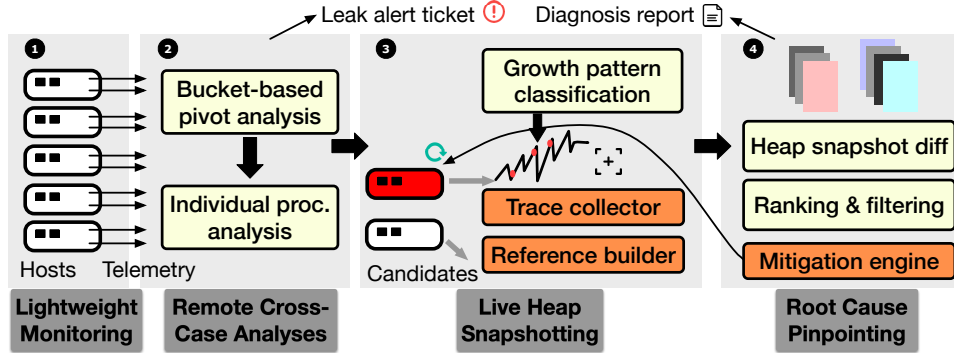


Figure 5.2: Workflow of the RESIN system.

different hosts using a bucketization-pivot scheme. If a bucket is suspected of leaking, RESIN triggers an analysis on the process instances from that bucket. After the two steps identify a highly suspicious software component, RESIN automatically generates an alert ticket for that component along with a list of leaking process instances belonging to that component. Meanwhile, RESIN performs live heap snapshotting (③) for the suspected processes. RESIN carefully chooses the snapshotting time using a growth pattern based algorithm to ensure the collected snapshots would be helpful. RESIN also samples normal processes to take regular heap snapshots and build a reference database. After generating multiple heap snapshots, RESIN tries to pinpoint the root causes (④) by running a diagnosis algorithm on the snapshots. The analysis report will be attached to the alert ticket thread to assist developers. Finally, RESIN automatically mitigates the leaking processes.

5.3.2 Design of Leak Detection

In this section, we describe the RESIN’s design for detecting memory leaks. In existing literature, the term “detection” refers to detect both (i) if a program or a process has a leak, and (ii) the bug in code or leaked objects. RESIN separates these as two tasks and uses the term detection to refer to (i) specifically. The diagnosis

component (Section 5.3.3) targets (ii).

5.3.2.1 Challenges

RESIN needs to address several challenges. First, cloud infrastructure software has highly noisy memory usage due to changing workloads and interference in the environment. Using static thresholds would generate many false positives. Standard anomaly detection algorithms [191, 192] do not work well either, because it is common for a component to exhibit memory usage spikes that are not leaks but legitimate increases in handling certain workloads.

Second, memory leaks in production systems are usually fail-slow faults [82] that last days, weeks, to even months (rapid leaks are likely caught in testing or deployment). Inspecting memory usage in a short time window would miss these slow leaks. It is necessary but challenging to capture gradual changes over a long period and still raise timely alerts.

Third, given the scale of Azure, collecting fine-grained data for a long time is impractical because of storage and overhead concerns. Therefore, RESIN can only collect limited, coarse-grained data and must work well under this constraint. Still, even with coarse-grained signals, the data volume is enormous. The detection algorithms must run efficiently.

5.3.2.2 Lightweight Memory Usage Monitoring

RESIN deploys a privileged monitoring agent on each host (Figure 5.3). This agent communicates with the host OS to track memory usage. It collects both kernel memory usage and per-process memory usage. The kernel usage is obtained from a pool monitor kernel module (PoolMon), and includes the usages of non-paged memory pool

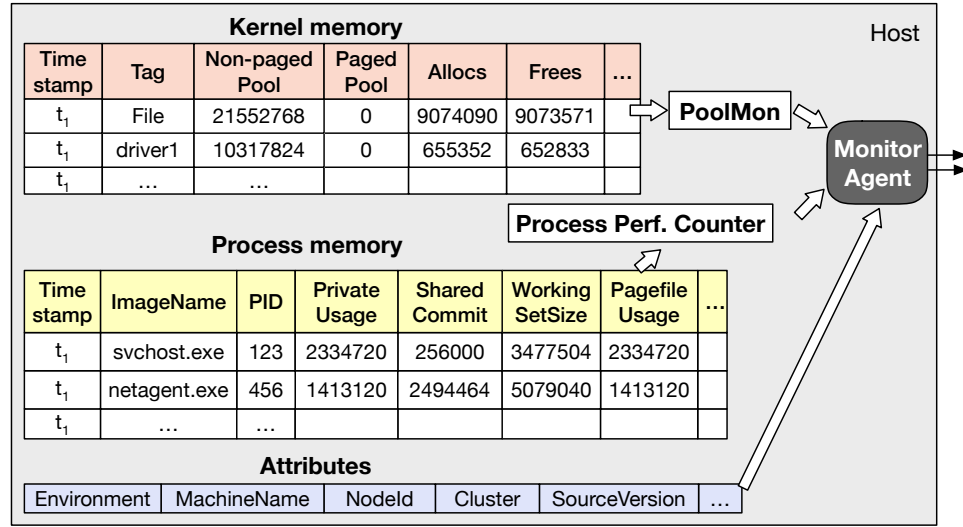


Figure 5.3: Monitor agent in each node collecting memory usage data.

and paged memory pool for each *tag*. The tag is passed as an argument by the callers of the kernel allocation API [168] and represents a sub-system that has requested memory from the kernel allocator, *e.g.*, the file system, a driver. The per-process usage is obtained by querying the per-process performance counters from the host. It includes breakdowns of a process's memory, such as the private commit, working set size, paging file usage, *etc.*

We collect the memory usage breakdowns and tags instead of simply a single total memory usage metric, because mixing different memory usage sources can introduce noises and miss important changes. For example, a 20 MB increase can be a leak for a driver but may be negligible for another component. Reporting the specific memory portion or tag that is leaking helps developers localize the buggy code. The breakdowns also help RESIN take more effective mitigation actions.

In addition to memory usages, the monitoring agent also records attributes such as the software version, hardware generations, node id, and cluster id. The attributes are used during leak detection analyses to increase accuracy. RESIN includes the

common attributes of the leaked process in the detection report to give developers troubleshooting hints.

The monitoring agent is scheduled to run every 5 minutes. However, the data points from different hosts may not be perfectly synchronized. Some special events in a host such as node reboots also introduce missing or invalid data. Therefore, RESIN aggregates the time-series data into hourly granularity by removing extreme outliers and computing the mean of the remaining data points. This pre-processing step reduces the noises as well as the data volume. Using an hourly window is not too coarse-grained because most software components in cloud infrastructure are long running, and production leaks typically occur in a large time scale.

5.3.2.3 Detection Algorithms

RESIN uses a two-level scheme to detect memory leak symptoms: a global bucketing-based pivot analysis to identify suspicious components, and a local individual process leak detection to identify leaking processes. The detection output includes the suspected component, the list of top leaking processes of that component, the leak start and end times, severity scores, *etc.* The detection algorithms are language agnostic.

Bucketization-based Pivot Analysis. To address the challenges described in Section 5.3.2.1, our insight is that we should inspect at the *component* granularity across processes. This is because although an individual process’ memory usage is influenced by workloads and highly noisy, the noises can be “canceled out” *en masse*. For a normal component, its process instances on different hosts may experience different workload effect at any time slice. But for a leaky component, the memory leak must be caused by some buggy release. Therefore, its processes should exhibit some

global trend at certain time slices despite the workload effect.

Based on this insight, we design a simple yet robust bucketization-based pivot detection scheme (Figure 5.4). Each circle represents one process. Shaded circle represents a process moving to another bucket. RESIN first groups the raw memory usage telemetry data into a number of buckets. In our implementation, we use 20 buckets (50 MB, 100 MB, 200 MB, ..., 40 GB, 50 GB). RESIN then applies pivoting to the data with a unique attribute tuple as the index and memory usage bucket as columns. The attribute tuple is (ProcessImageName, ServiceName) for user-level software, and (TagName, PoolType) for kernel subsystems, where Type is paged memory or non-paged memory. The aggregation function is the count of distinct nodes. Thus, each summary cell represents the number of nodes that have running processes with a particular attribute tuple and these processes' memory usages fall into the specific bucket. RESIN computes the summary periodically and incrementally for data in each time interval. The results are saved into a database table.

We basically transform the memory usage data into summary about numbers of nodes in different buckets, which can more robustly represent the trends and tolerate noises due to workload effect (*e.g.*, the non-leaky component in Figure 5.4). RESIN then runs anomaly detection on the time-series data of each bucket for each component. It uses the most recent time period of summary data (default 15 days), with the first 2/3 portion as the baseline and the remaining data points as the test. If a bucket's test period has data points that exceed the $\mu + 3\sigma$ of the baseline data (μ and σ represent the mean and standard deviation of the distribution), it is considered to be an anomaly. The start time and end time in the test period when the node count becomes the outlier are recorded.

One caveat is that if many processes of a component experience a sudden *drop* in

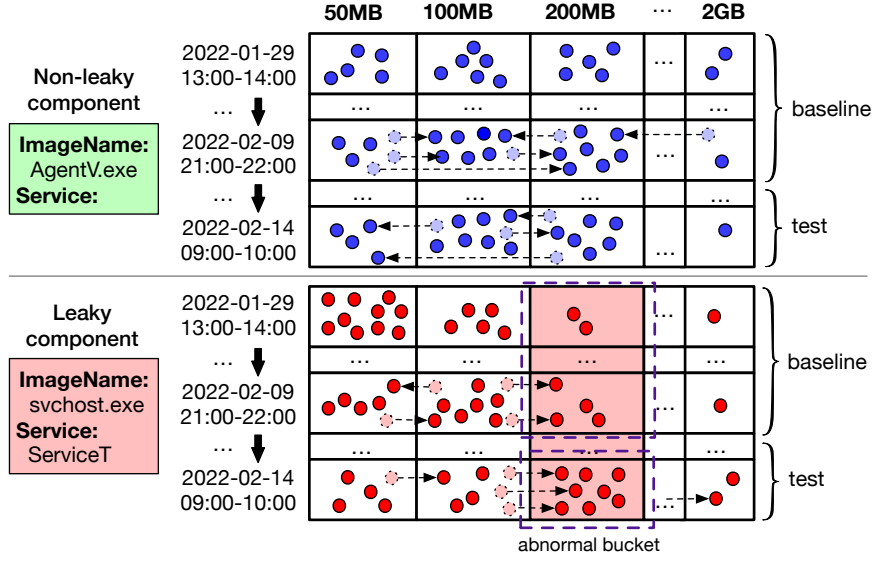


Figure 5.4: The memory usage grouped into buckets and pivot by image name, service, and bucket size.

memory usage, the node count would shift from a higher bucket to a lower bucket. But the lower bucket's node count significant increase is not an anomaly. To handle such scenarios, RESIN calculates cumulative bucket values during the anomaly detection. In other words, if the 100 MB bucket has a node count of n , it means there are n nodes that have the particular processes with memory usage equal to or larger than 100 MB, including processes (if any) that fall into the 200 MB bucket. In this way, a significant increase in a bucket almost always suggests an anomaly.

The bucketization approach also helps address the computation challenges. Before introducing this approach, it can take RESIN more than one day to run anomaly detection on the enormous data points from millions of nodes. After the pivot summary, RESIN only needs to run anomaly detection for the time-series data in each bucket, which can finish in less than one hour for all data (even without parallelization).

RESIN calculates a severity score for each bucket based on the deviations and

node count in the bucket. It considers a component is leaking based on a $\langle size_mb, score \rangle$ threshold: if a bucket is of size equal to or larger than $size_mb$ and its severity score exceeds $score$. RESIN generates intermediate reports for the abnormal buckets. It de-duplicates the intermediate reports by only keeping the one for the largest bucket of a unique attribute tuple, and generates a ticket for that report.

Localizing Individual Processes. The bucketization pivot analysis works at the component granularity. RESIN uses a second-level detection scheme that works at the process¹ granularity. The motivation for this scheme is that a component has many process instances. It is important to localize the truly leaking processes in the alerting bucket. If we simply include all the processes in the abnormal bucket, developers can waste significant effort investigating innocent processes that fall into that bucket by coincidence.

The second-level detection scheme computes the leak likelihood and severity for a process based on its memory usage. RESIN uses all the memory usage data of a component in the most recent month to train two parametric models: (i) the *absolute usage model* and (ii) the *usage difference model*. Since different clusters and regions can exhibit drastically different characteristics, the tool builds separate models for each combination of region name and cluster type for a component.

Let $U_c(n_i, t_j)$ denote the memory usage value for a process of component c on node n_i at time t_j . RESIN assumes *absolute usage* $U_c(n_i, t_j)$ follows a Gaussian distribution $\mathcal{N}(\mu_1, \sigma_1^2)$ and fits the memory usage data by calculating the maximum likelihood estimators for μ_1 and σ_1 . The absolute memory usage values can be severely

¹Here we use the term “process” to also include a running instance of a kernel subsystem in a particular host for kernel memory leak detection.

Algorithm 3: Moving suspicious interval algorithm

Input: $U_c(n_i, t)$: time-series memory usage for node n_i of component c ; $\mathcal{N}(\mu_2, \sigma_2^2)$: offline usage difference model for component c .
Output: T_0, T_1 : leak start and end time; no leak if $T_0 == T_1$. N_{inc} : number of increasing data points
 $t_n \leftarrow \max(t) \text{ in } U_c(n_i, t), N_{inc} \leftarrow 0$
 $T_0 \leftarrow t_1, T_1 \leftarrow t_1$
for $j \leftarrow 2$ **to** n **do**
 $T_1 \leftarrow t_j$
 $\Delta U_c(n_i, t_j) \leftarrow U_c(n_i, t_j) - U_c(n_i, t_{j-1})$
 if $IsOutlier(\Delta U_c, \mathcal{N}) \parallel U_c(n_i, t_j) < U_c(n_i, T_0) \parallel N_{inc}/n < \varepsilon$ **then**
 $T_0 \leftarrow t_j$
 if $T_0 == T_1$ **then**
 $N_{inc} \leftarrow 0$ /* empty interval, no leak, reset */
 else if $IsLarger(U_c(n_i, t_j), U_c(n_i, t_{j-1}), \mathcal{N})$ **then**
 $N_{inc} \leftarrow N_{inc} + 1$ /* a new increasing data point */
end
return T_0, T_1, N_{inc}

distorted by occasional events such as VM creations. To account for such events, we consider the differential memory usage, *i.e.*, $\Delta U_c(n_i, t_j) = U_c(n_i, t_j) - U_c(n_i, t_{j-1})$. Based on our observations, when noisy events such as VM creations occur, $\Delta U_c(n_i, t_j)$ usually significantly deviates from its normal range. Thus, RESIN also builds a parametric Gaussian distribution $\mathcal{N}(\mu_2, \sigma_2^2)$ model for *usage difference* $\Delta U_c(n_i, t_j)$ and calculates the μ_2 and σ_2 .

With the offline models, RESIN uses a *moving suspicious interval* algorithm (Algorithm 3) to examine a suspected process' memory usage in *real time*. This algorithm works by keeping a *suspicious leak time interval* $[T_0, T_1]$. The basic idea is to assume the leak still continues at the end of the time series and try to find the earliest time the leak trend starts by skipping over low-confidence points. This interval is initialized as $[t_1, t_1]$ upon reading the first data point in a time series. At the j -th step, RESIN reads $U_c(n_i, t_j)$, calculates the $\Delta U_c(n_i, t_j)$, and adjusts the time interval by moving T_1 and update T_0 adaptively. If $\Delta U_c(n_i, t_j)$ has a significant increase or drop (based on the 3-sigma rule for μ_2 and σ_2), T_0 is updated to t_j because the system

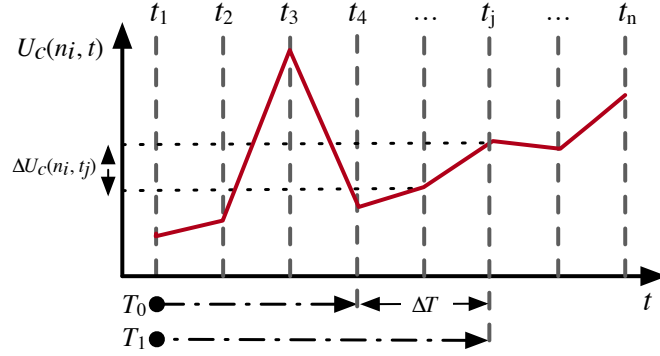


Figure 5.5: Applying the moving suspicious interval algorithm.

status is likely changed by some event. If $U_c(n_i, t_j)$ is lower than $U_c(n_i, T_0)$ or there are few increasing points in the current interval, T_0 is also updated to t_j because a leaking trend should have enough increasing values. For other situations, we keep the T_0 intact. The loop stops when T_1 hits the last time point t_n . If the final T_0 is equal to T_1 , this process is not considered as leaking.

Figure 5.5 shows an example of applying the algorithm. $[T_0, T_1]$ are initially set to $[t_1, t_1]$. When T_1 is set to t_2 , $\Delta U_c(n_i, t_j)$ is positive thus we keep T_0 unchanged and continue to move T_1 forward. When T_1 is set to t_3 , $\Delta U_c(n_i, t_j)$ is an outlier in the offline model $(\mathcal{N}, \mu_2, \sigma_2^2)$, which we consider an occasional event instead of a leak. We reset T_0 to t_3 accordingly. When T_1 is set to t_4 , $U_c(n_i, t_j)$ is significantly lower than $U_c(n_i, T_0)$ thus we also reset T_0 to t_4 . After t_4 we did not encounter scenarios to reset T_0 (the memory usage drops slightly later but it is unnecessary to reset for such cases), so eventually T_1 reaches the end t_n , and $[T_0, T_1]$ is $[t_4, t_n]$.

RESIN calculates a severity score (Equation 5.1) for a process to indicate its leak probability and impact. Several factors are considered, including the normalized memory usage difference ($\Delta U_c = U_c(n_i, t_n) - U_c(n_i, t_1)$), the length of the suspicious leak interval ($\Delta T = T_1 - T_0$ in the unit of month), the increasing rate (number of increasing data points over the number of all data points), and the final memory

usage at t_n .

$$SevScore = \frac{\Delta U_c}{\sigma_1} + \frac{N_{inc}}{n} + \Delta T + \frac{1}{1 + e^{-U_c(n_i, t_n)/\mu_1}} \quad (5.1)$$

For efficiency, the above analyses are run proactively. When the bucketization-based pivot detection step identifies a leaking bucket, RESIN triggers the individual process analysis for the processes in the bucket and can usually inspect the results without waiting. It outputs the suspected leaking processes, the leak start and end time, and the severity scores.

5.3.3 Diagnosis of Detected Leaks

Only detecting a leak is not enough. Without sufficient evidence and diagnosis support, developers are likely stuck in confirming and diagnosing the issue. RESIN designs a solution that automatically takes *live* heap snapshots and analyzes the snapshots to pinpoint the root cause of a detected leak.

5.3.3.1 Background: Heap Snapshot

RESIN provides diagnosis information at stack trace granularity. In our experience, if developers are presented with a stack trace containing the problematic allocations, they can often quickly debug the issue. RESIN leverages the Windows heap manager's snapshot capability to perform live profiling. The heap manager exposes APIs such as `HeapAlloc`, `HeapReAlloc`, and `HeapFree`, which are used by applications and C/C++ runtime to allocate heap objects. Thus the heap manager has the ability to collect the heap allocation sizes and stack traces.

RESIN uses the Windows Performance Recorder [211] to notify the kernel to start tracing heap allocations, typically for a specific process ID and occasionally

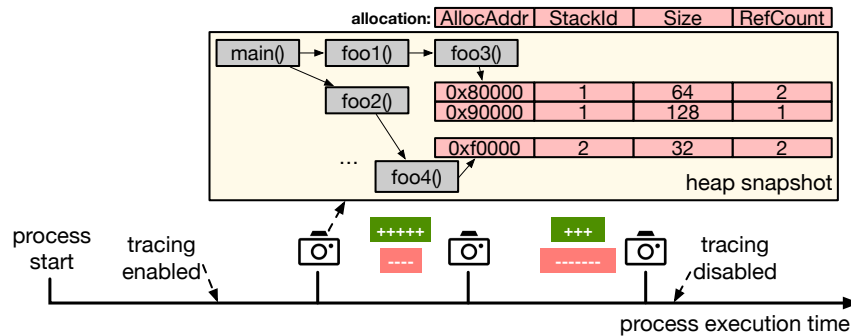


Figure 5.6: Periodic heap snapshot collection.

for an image name (which enables tracing for all processes with the image). Then RESIN instructs the heap manager to take a snapshot at a certain time. Our current heap snapshotting mainly focuses on C/C++, which are the primary language choices for host software on Azure. Extending to other languages would take extra effort but are still straightforward, as their runtime typically already provides the functionality to capture allocation events.

To minimize overhead, the heap manager only stores limited information in each snapshot. Specifically, it stores (1) the stack trace and size for each *active* allocation after the tracing was enabled (if an allocation has been freed, no information is stored), (2) the total allocation sizes for each unique stack trace, and (3) the number of times a unique stack trace is invoked. It does *not* store more detailed information such as the allocation time or a pointer graph.

The information in a single snapshot is usually too noisy, as it includes all active allocations from the tracing start to the snapshot point. To get more accurate information for a time window, RESIN periodically takes *multiple* heap snapshots (Figure 5.6) to increase the chance of capturing truly leaking allocations between snapshots. RESIN uploads the snapshot files to a remote storage service. The diagnosis engine uses these snapshots to deduce the leaking allocation points.

5.3.3.2 Choosing Candidate Hosts to Profile

Picking the right hosts to take heap snapshot is vital for diagnosis effectiveness. Because heap snapshot incurs overhead, RESIN cannot afford to enable snapshot on all hosts containing the leaking processes the detection engine outputs. Simply choosing the hosts randomly is not a good strategy either, because the workloads on different hosts vary widely. For the same leak bug, it can exhibit in quite different patterns on different hosts. Thus, we may choose a candidate host in which the buggy allocations are triggered rarely.

We rank the candidate hosts in the suspected list based on three factors: 1) *severity*: choose processes with higher severity scores as described in Section 5.3.2.3, since more obvious symptoms suggest a better chance to be diagnosed; 2) *noisiness*: choose processes with a clearer growth pattern, which we will discuss in more detail in Section 5.3.3.3; 3) *impact*: choose hosts that have fewer user activities to minimize impact of profiling events. By default RESIN triggers snapshot collection for the top three hosts in the list in case the collection fails unexpectedly (*e.g.*, due to target process restart).

5.3.3.3 Deciding Trace Collection Strategy

With the candidate hosts selected, the next step is to decide if a *new* leak happens in the most recent snapshot interval and whether to take the snapshots. This step is different from the analysis in Section 5.3.2.3, which only finds leaking processes in *past* time. The decision making has two main challenges.

First, many production leaks are only triggered by specific events. Some leaks only occur once in several days. If we take snapshots at other times, the collected traces would not be helpful. To ensure rare leaks are captured, RESIN attaches the

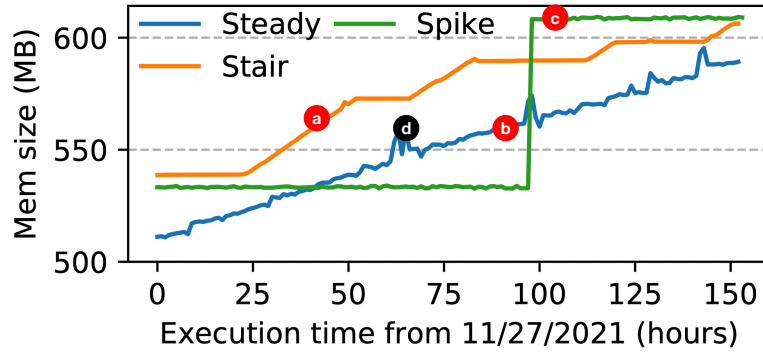


Figure 5.7: Memory growth patterns and completion point choices. Each data point is real memory usage from production processes.

profiling workflow to the process for a long time and periodically (every half hour) takes snapshots in hope of capturing the leak. However, we cannot afford to keep uploading snapshots due to storage and overhead concerns. RESIN addresses the challenge with a *long-term, trigger-based* strategy: it uses a circular buffer that only keeps the most recently taken snapshots, and completes tracing once certain trigger is met.

Second, how to decide when the trace collection should complete, *i.e.*, the trigger. At the completion time, we should ideally (1) have snapshot(s) containing the buggy allocation; (2) have snapshot(s) for non-leaking scenarios; (3) minimize noisy allocations in the snapshot(s). One potential trigger is to complete the collection once the memory usage difference exceeds some threshold. This trigger can easily complete the tracing prematurely (fails to capture the buggy allocation) due to a legitimate memory usage spike, and/or produces snapshots that have many noisy allocations and mislead diagnosis.

To gain some insights on how to choose the triggers, we study the memory usage data of 51 real leak cases. Interestingly, most cases fall into three common patterns (and a mixture of them). Additionally, one leaking process in a specific host often

Pattern	Characteristics	Completion Trigger
Steady	Almost linear growth	$R_j^2 > \lambda_a$.
Stair	Steady growing and flat curves alternately appears.	$ 1 - slope_j / slope_k < \lambda_b$ && $R_j^2 < \lambda_c$
Spike	A few large allocations in a short period of time	$\Delta U_c(n_i, t_j) / \Delta U_c(n_i, t_k) > \lambda_d$

Table 5.1: Leak patterns, characteristics and their completion triggers.

has a *consistent* pattern. In 63% of the cases, the leaking process shows a *steady* pattern. One example is the bug shown in Figure 5.1, in which the leak exists in periodical update tasks. The other two common patterns are *stair* (the memory usage occasionally grows only when the procedure containing leaks is activated) and *spike* (the leak only occurs once in a while due to rare events). Each pattern has its unique usage growth characteristics and clear completion point candidates ①, ②, ③ shown in Figure 5.7. ① is not a good completion point as it is right after a period of some noises.

Guided by the findings, RESIN takes a pattern-based approach to decide the trace completion triggers. It uses the target process' memory usage data in the most recent week and classifies it into one of the three patterns. Specifically, RESIN first identifies nearly flat segments in the time-series data and removes them. It then performs linear regression on the remaining growing segments and outputs results including the slope $slope_k$, coefficient of determination R^2 , and absolute memory usage increase $\Delta U_c(n_i, t_k)$. If the data contains no flat segments, RESIN marks it as a *steady* pattern. If the data has flat segments in between growing segments, it is marked as a *stair* pattern. If a large increase in memory usage only occurs in a few data points, it is marked as a *spike* pattern.

After the pattern is classified, the workflow starts to monitor and analyze the

Algorithm 4: Heap snapshot diagnosis algorithm

Input: A_{n-1}, A_n : sets of allocations in two heap snapshots, S^r : a list of outstanding stacks from reference hosts, $pattern$: classified pattern, $estimate_leak$: upper bound of estimated leaking size

Output: S^o : a list of top N stack traces that likely caused leaks

```
 $S^o \leftarrow [], S^{diff} \leftarrow []$   
 $S^n \leftarrow A_n.groupby(alloc \Rightarrow alloc.stackid)$   
 $S^{n-1} \leftarrow A_{n-1}.groupBy(alloc \Rightarrow alloc.stackid)$   
foreach  $stack \in S^n$  do  
  if  $stack \in S^{n-1}$  then  $A^{diff} \leftarrow S^n[stack.id] \setminus S^{n-1}[stack.id]$   
  else  $A^{diff} \leftarrow S^n[stack.id]$   
  if  $A^{diff} \neq \emptyset$  then  
    foreach  $a \in A^{diff}$  do  $stack.size \leftarrow stack.size + a.size$   
     $S^{diff}.add(stack)$   
  end  
end  
 $S^{diff}.orderBy(stack \Rightarrow stack.size)$   
if  $pattern \neq SPIKE$  then  
   $S^{diff}.removeAll(stack \Rightarrow stack.size > estimate\_leak)$   
end  
 $S^{diff}.removeAll(stack \Rightarrow stack \in S^r)$  /* filter references */  
 $S^o \leftarrow S^{diff}.top(N)$  /* only keep top N stack traces */  
return  $S^o$ 
```

recent memory usage. We compute the $slope_j$, R_j^2 , and $\Delta U_c(n_i, t_j)$ for the most recent six hours and check the pattern's completion trigger based on rules listed in Table 5.1 ($\lambda_a, \lambda_b, \lambda_c$ and λ_d are set to 0.8, 0.1, 0.1, and 0.5, respectively). Once the trigger is satisfied, RESIN stops tracing and uploads the trace file that contains the most recent few snapshots. It ensures each trace has at least three snapshots.

5.3.3.4 Collecting Reference Snapshots

One challenge in using snapshots for diagnosis is the presence of many noisy but benign allocations. Even with multiple snapshots, they may remain active and mislead the diagnosis. RESIN collects *reference snapshots* to address this challenge.

For the reference snapshots to be useful, they should be comparable to the snapshots from the leaking process. A poor choice of a reference snapshot may be even

counter-productive and filter out the culprit allocations. RESIN uses a periodical *fingerprinting* process to build reference snapshots. It randomly samples hosts for common leaking services to take heap snapshots. We currently define the fingerprints to be the attribute tuple (`cluster_id`, OS version, service version, date). This is based on our observations on the locality of memory leaks. These snapshots are saved in a reference database and cleaned up when they become stale.

At the diagnosis stage, after RESIN chooses the candidate leaking hosts to profile (Section 5.3.3.2), RESIN checks if the database already has reference snapshots with similar fingerprints. If not, RESIN triggers reference collection. It first scans the qualified hosts (not in the detection engine’s suspicious list and have similar fingerprints to the leaking hosts) and samples a few that have active memory activities and modest memory usage. Then RESIN applies the growth pattern analysis (Section 5.3.3.3) to check if this host is leaking. If not, it takes snapshots and uploads the traces to the reference database.

5.3.3.5 Trace Analyses for Diagnosis

The next step is to analyze the collected snapshots to output the root cause stack traces. The challenge is to handle many noisy allocations and localize the buggy allocations.

RESIN designs a diagnosis algorithm listed in Algorithm 4. The inputs are the allocations from the two most recent snapshots of a trace file (A_{n-1} , A_n), stack traces from reference snapshots, and the estimated leaked size upper bound calculated in the pattern analysis. For the *steady* and *stair* patterns, we estimate the leaked size upper bound by multiplying the slope with the time interval of the growing segments and a coefficient (by default 2). The goal is to find the stack traces that allocate objects of

sizes closest to the estimated leak.

The diagnosis engine first groups allocations in A_{n-1} , A_n by the stack trace id and get two maps S^n and S^{n-1} . Each map value is all the allocations that come from a stack trace. It then traverses each stack trace in S^n to calculate the aggregated allocation size. The engine then identifies stack traces that contain unique allocations, and ranks these traces based by their allocated object sizes. Stack traces allocating sizes larger than the estimated leak size are likely noises and thus removed. Finally, the diagnosis engine cross-checks the reference snapshots to filter out benign stack traces. If the output list is empty, the engine repeats the analysis for the next snapshot pairs (A_{n-2}, A_{n-1}) , *etc.*

5.3.4 Mitigating Leaks

When a memory leak is detected, it can take time for developers to come up with and deploy the bug fix. To avoid further customer impact, RESIN attempts to automatically mitigate the detected leak issues. Depending on the nature of the memory leak, mitigation can be done in several ways. Rebooting the host OS in general can mitigate all kinds of leaks. However, this is costly and potentially causes VM downtime.

RESIN leverages the results from its detection engine and uses a rule-based decision tree to choose a mitigation action that can minimize the impact. If the memory leak is localized to a single process or Windows service, and this process or service is not required to be always alive to provide services to customers, RESIN attempts the lightest mitigation by simply restarting the process or Windows service.

For some processes, the mitigation requires additional steps. RESIN allows component teams to define custom scripts and invoking conditions. If the leak is located in buggy drivers, RESIN unloads and reloads the driver to mitigate the issue.

For safety, RESIN uses allowlists for each action category to make sure auto-mitigation is not misused. It defines an initial allowlist for the processes and drivers that are known to be safe to restart. A feature team can opt in auto-mitigation by adding the name of the process or tag to the allowlist.

For leaks in the OS kernel memory such as I/O request objects and file objects, if the detection engine can attribute the leak to a process or a service, RESIN attempts to restart the culprit process or service. This action is usually effective because it allows the leaked kernel objects to be properly freed without the need to reboot the OS.

OS reboot will resolve any software memory leak but takes a much longer time and can cause VM downtime. Thus, it is the last resort when a leak cannot be mitigated by the above actions or the name is not in the allowlist. RESIN checks if the host is empty and does the OS reboot if so. Empty hosts could also leak memory due to past user activities or current background processes. For a non-empty host, RESIN first performs live VM migrations [48]. Then it attempts a kernel soft reboot, which skips hardware initialization. If the soft reboot is ineffective, a full OS reboot is performed.

To minimize the impact of mitigation actions, RESIN closely monitors the leaking hosts. It prioritizes the actions on 1) nodes that fire low-memory related events, such as E2004 (low virtual memory) from the Windows resource exhaustion detector, E3122 (not enough memory to start VM) from Hyper-V; 2) nodes in regions with capacity issues; 3) nodes with host memory reservation overage; 4) nodes ordered by the leak size, leaking rate, and the predicted time-to-failure.

RESIN stops applying mitigation actions to a target when the detection engine

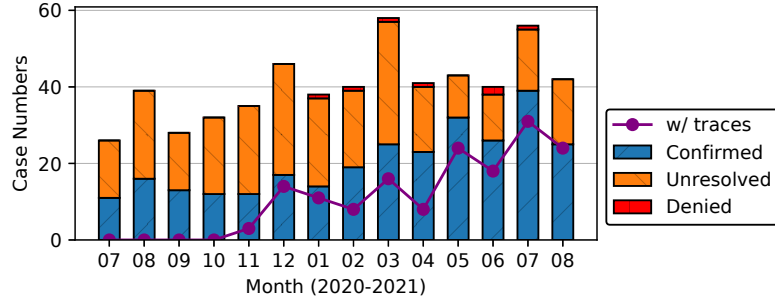


Figure 5.8: Memory leak cases RESIN detected and reported.

no longer considers the target leaking. This typically occurs *without* manual intervention. For example, after developers identify the root cause and apply a fix, the leak symptom disappears, so RESIN stops the mitigation. Sometimes, after a mitigation action, the leak symptom no longer re-appears, which naturally stops further mitigation.

RESIN also coordinates with its diagnosis engine (Section 5.3.3) in performing the mitigation actions. If the diagnosis engine plans to or is taking heap snapshots for a candidate host, RESIN defers the mitigation actions to avoid losing the critical opportunities for capturing the leaking allocations.

5.4 Evaluation

Our evaluation answers several questions: (1) how effective is RESIN in detecting memory leaks? (2) how accurate is the detection? (3) can RESIN help developers diagnose and mitigate leaks? (4) what is the overhead of trace collection?

5.4.1 Deployment Status and Scale

RESIN has been running in production in Azure since late 2018. It covers millions of hosts, over 600 different host processes and over 800 different kernel pool tags daily. The detection engine in RESIN analyzes more than 10 TB memory usage data every

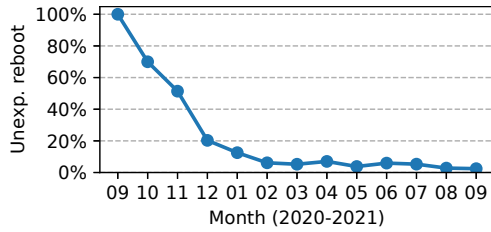


Figure 5.9: Unexpected VM reboot.

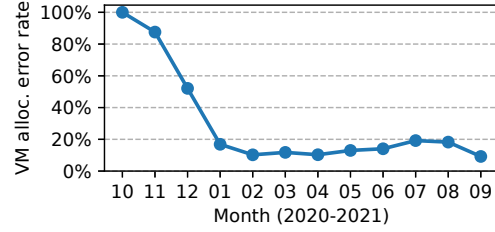


Figure 5.10: VM allocation error rate.

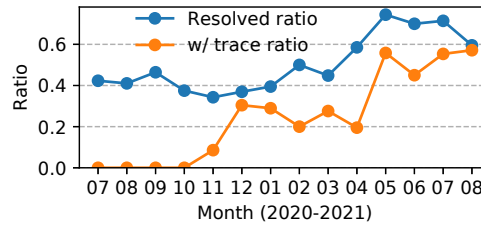


Figure 5.11: Ticket resolved and w/ trace ratio.

day. The diagnosis module collects 56 trace files on average (10–200MB) daily. Every month, the mitigation engine performs a median of 1,592 process restarts, 1,290 kernel soft reboots, and 4,649 node reboots.

5.4.2 Detecting Production Memory Leaks

Azure has various solutions that help eliminate memory leak bugs before production, including code reviews, static bug finding tools, testing, and safe deployment policies. As a result, only complex memory leak bugs occasionally escape these solutions. RESIN serves as the last defense to effectively catch these bugs in production.

Figure 5.8 shows the memory leak tickets RESIN reported in Azure from July 2020 to August 2021. Overall, RESIN reported 564 tickets in 14 months, among which developers explicitly resolved 291 tickets.

5.4.3 End-to-End Impact

The end-to-end benefits brought by RESIN are clearly demonstrated by two key metrics: (1) *VM unexpected reboot*²: the average number of reboots per one hundred thousand hosts per day due to low memory; (2) *VM allocation error*: the ratio of erroneous VM allocation requests due to low memory.

As shown in Figures 5.9 and 5.10 (data is normalized for confidentiality), the improvement RESIN provides is significant. Both metrics show large decreases: VM reboots are reduced by $41\times$ from September 2020 to September 2021, and allocation error rates are reduced by $10\times$ from October 2020 to September 2021. Note that these key metrics have been continuously dropping before the starting points in Figures 5.9 and 5.10. We omit to plot these earlier points because the raw data are no longer in the database due to data retention policies.

The improvement also shows in the reduction of service incidents. In 2020 and 2021, no severe outages in Azure were caused by memory leaks (such outages occurred previously).

5.4.4 Effectiveness of Detection

Detecting memory leaks in a complex, frequently changing cloud infrastructure like Azure is challenging. RESIN aims to minimize the false positives and false negatives.

Precision To evaluate the detection accuracy, we count how many cases RESIN reported are flagged by developers as false alarms. Overall, RESIN only incurs 7 false positives out of 291 resolved cases in 14 months.

There are two common patterns of false positives: (i) after a component adds

²In this chapter, we only count those VM unexpected reboots and allocation errors caused by low memory and memory exhaustion.

a new feature that consumes significantly more memory; (ii) after a configuration change, *e.g.*, “*we are collecting and correlating lot more counters and # of disks per node have also increased*”. The memory usages in these cases typically stabilize and become new baselines, which RESIN automatically picks up without requiring adjustment.

Recall We count how many cases RESIN misses or fails to detect in time. The criterion is a memory leak causing noticeable service impact and getting reported by users, developers, or other monitoring services before RESIN detects it. We search keywords including “memory leak” and “leak” on Azure’s issue tracker used by all teams. We compare them with tickets automatically created by RESIN. Overall the false negatives are few: only 4 cases are not on RESIN’s ticket list.

We also inspect the reason for each case. One case in July 2020 was caught by us manually when analyzing a heap snapshot trace, before the issue triggered RESIN’s alert. In other two cases, the leak impact was not significant, but developers found the leaks when they were closely monitoring their new deployment. The last case was both found by developers and RESIN, but developers found the issue faster. This happened because the issue manifested itself earlier on a testing cluster that had a different workload from the production clusters RESIN monitored; developers were also using an aggressive threshold in that cluster to expose potential issues.

Timeliness It can be difficult to determine the exact starting time of a leak and hence the exact detection delay. We observe that RESIN’s detection typically occurs within two hours of a leak’s clear manifestation. In general, a too-long detection delay would be reflected in a false negative since developers or other monitoring tools would detect the issue earlier.

Resolution rate Not all cases are eventually resolved by developers. On average, RESIN reports a resolution rate of 52% within reported date range. We believe this resolution rate is underestimated compared to actual responsive rate: in many cases, developers took actions but did not update the open tickets (only tickets tagged with a high severity level are mandatory to resolve according to ticket platform’s policy). “Unresolved” also does not necessarily mean the ticket is unimportant. We observed that some teams tend to have lower response rates, likely due to their limited resources and the overwhelming number of urgent tickets.

This result is also influenced by our design goal. RESIN is designed to catch memory leaks early before the leak escalates to catastrophic issues. RESIN further provides automatic leak mitigation. Thus, a could-have-been-severe leak would appear to be low-risk. When developers prioritize resolving high-impact tickets, it adversely affects the resolution rate for our leak tickets. Due to the large volume of reported cases from various groups, we could not afford to inspect all unresolved tickets and check with their owners.

5.4.5 Effectiveness of Diagnosis

The diagnosis module of RESIN in total collects traces and generates reports for 157 cases from July 2020 to August 2021 (Figure 5.8). For tickets related to kernel leaks or clusters with legacy OSes, RESIN is unable to collect traces. Before November 2020 the diagnosis module was in trial runs and diagnosis reports were not appended to tickets. We gradually enabled it for more clusters after the trial run was over.

Figure 5.11 shows how the ratio of tickets with collected traces (and diagnosis reports) increases, which correlates with the improvements on the ratio of resolved tickets.

Usefulness To evaluate the usefulness of the diagnosis reports RESIN generates, we randomly sample 14 issue tickets and closely follow up with the developers (all cases are eventually resolved and fixed). In 11 cases (79%), developers directly use the diagnosis reports to pinpoint root causes. Among them, in 5 cases the bug fix is in the same function in the allocation stack; in 5 cases the allocation stack and bug fix are in the same source file; in only 1 case the allocation stack and bug fix are in different components. Out of the other three cases, two are solved by memory dumps because the developers are quite experienced: upon seeing the sizes of leaked memory objects, they immediately realized which function the leaks came from. In one case the traces captured in the production cluster are not particularly useful. Instead, developers successfully captured some snapshots consisting of leaking stack on their own testing cluster.

Feedback Over time, developers build up high confidence in RESIN. We receive many pieces of positive feedback:

“(The result is..) incredibly useful. The information I had was enough.”

“Thanks for pinpointing out the memory leak that we had been trying so hard to find over the past few days.”

“Stack trace was sufficient for debugging this, it included the API call that was problematic.”

Case studies We share two representative cases. The first case occurs in ServiceH³. This process’ memory usage keeps increasing and gets restarted every few days. The diagnosis module in RESIN collects heap snapshots and pinpoints the root cause stack trace. After the diagnosis report is attached to the ticket, developers confirm

³The service names are anonymized for reasons of confidentiality.

```

1 virtual void AddDsmsCertificate(CertificateStore& ... {
2 - for (; certHead != nullptr; certHead = certHead->Next) {
3 + for (auto currentCert = certHead; currentCert != nullptr;
4 +   currentCert = currentCert->Next) {
5 - if (certHead->Versions == nullptr)
6 + if (currentCert->Versions == nullptr)
7   continue;
8 - auto latest = certHead->Versions->Latest;
9 + auto latest = currentCert->Versions->Latest;
10  if (latest == nullptr)
11    continue;
...
20 freeCertLst(certHead)

```

Figure 5.12: The fix for ServiceH leak.

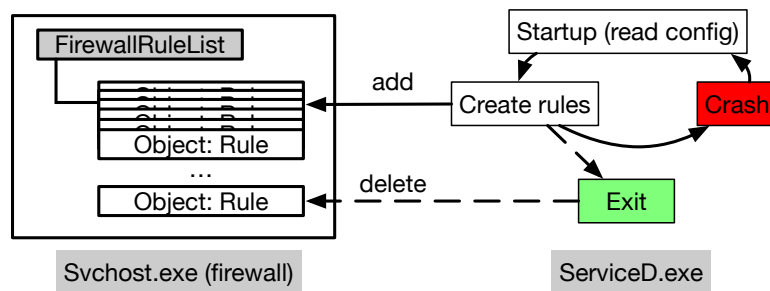


Figure 5.13: Contract violation induced leak on ServiceD.

and fix the issue in 3 hours.

In this case, the program uses a pointer to manage the list of certificates, and frees the pointer at the end of the function. However it also uses the pointer to traverse the list. In the end the pointer has moved and only a part of the list is freed (Figure 5.12). This is a day-0 bug introduced a long time ago, but is recently triggered due to added certificates to the machines.

The second case represents another common (6 out of 14 cases we studied) type of leaks in cloud infrastructure: leaks due to contract violations in cross-component interactions. After RESIN reports a firewall-related svchost is leaking, the diagnosis module collects traces and reports a function in the rule list adding procedures after analyses.

Developers do not find bugs in this specific function at first, but the report prompts them to check the firewall rule lists on these machines. They then find the rule lists

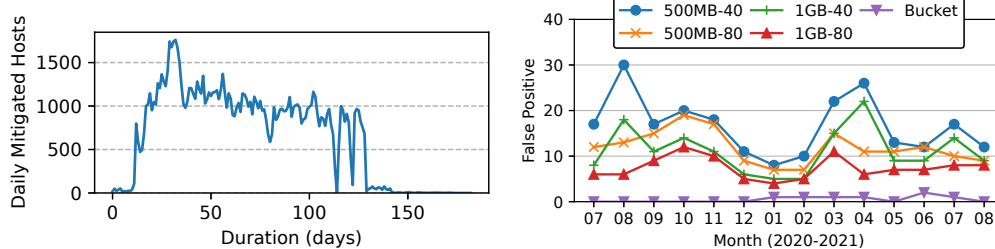


Figure 5.14: Mitigation for a leaking driver. **Figure 5.15:** False positive of detection algo.

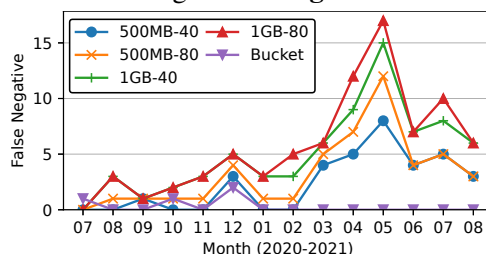


Figure 5.16: False negative of detection algo.

on these machines have been flooded with redundant rules. The reason is that the svchost process gets a firewall configuration from another program ServiceD. This program creates firewall rules at startup. Due to another bug, ServiceD keeps crashing, which causes it to miss deleting created rules and repeatedly recreate rules upon restarts (Figure 5.13). This in turn causes significant memory usage increases for the svchost program. Such a bug is hard to be detected statically.

Timeliness The diagnosis timeliness is also important to help developers. We measure the latencies of RESIN’s heap snapshot collection and analysis. The median trace collection time is 61 minutes. For more than 80% of cases, the collection finishes within 10 hours. Note that the trace collection time is influenced by when a leak recurs in a suspected process. If the leak is sporadic, RESIN has to wait until the symptom reappears to capture the snapshot. For trace analysis, the median latency of the analysis jobs is 10 minutes.

Mitigation	Count	50%	75%	90%	99%
Process restart	27,039	1.62	5.74	6.50	30.70
Kernel soft reboot	8,292	24.64	34.47	49.14	141.69
Node reboot	278,005	248.58	274.36	362.10	1382.61

Table 5.2: Single mitigation action execution time (seconds).

5.4.6 Effectiveness of Mitigation

Mitigation procedure duration on leaked services Figure 5.14 shows the number of mitigated nodes of a kernel leak due to a buggy driver. At first, RESIN applied mitigation actions on a few nodes per day to test possible side effects. Once the mitigation actions reached production, RESIN applied mitigation to at most around 2,000 nodes per day with some fluctuations. The mitigation action volume then gradually dropped as the fix was being rolled out. Eventually the volume dropped to a few nodes a day, which were primarily nodes that failed in driver upgrading or other fix actions.

Mitigation action duration on single host We collect the frequencies and durations of each mitigation action between July 2020 to September 2021. As Table 5.2 shows, process restart is the most lightweight mitigation action. In most cases, it finishes within 6.5 seconds. Kernel soft reboot is also fast and in most cases finishes in a minute. Node reboot takes a longer time, with a median time of 4.6 minutes.

5.4.7 Comparison of Different Algorithms

Bucketization-based detection We first compare our core detection algorithm, the bucketization-based pivot analysis, with the practice of static threshold-based memory usage monitoring. We use four threshold policies, *e.g.*, policy “500MB-40” means generating leak alerts if a service’s memory usage exceeds 500 MB on more

than 40 nodes. We apply these hard thresholds to historical data and count how many cases will be wrongly reported as leaking (false positive) and how many leaking cases will be missed (false negative).

Figures 5.15 and 5.16 show the results. Our algorithm performs the best: it has both the lowest false positives and the lowest false negatives. In comparison, for other policies, it is often a dilemma to balance precision and recall. For example, policy “1GB-80” has the lowest false positives among the baselines at the cost of having the highest false negatives.

Pattern-based collection We compare our pattern-based collection with random collection. The experiment is conducted on ServiceS, ServiceV, and ServiceW. They have ongoing memory leaks on some hosts. We randomly choose six hosts and apply pattern-based collection on three hosts and random collection on the other three hosts. For the random strategy, we implement a workflow that periodically collects snapshots with at least two snapshots and completes the trace collection with a probability $1/6$. We inspect the collected heap snapshot traces to see if the leaking allocation exists in the snapshot.

Our pattern-based collection successfully captures leaking allocation stacks for all three services. Interestingly, the root cause of ServiceW was still unknown at the time we conducted the experiment. RESIN successfully captures an outstanding allocation that contains the bug within a real-time event processing function. In comparison, random collection only captures the buggy allocation for ServiceS, which has a frequent leaking interval (less than 1 hour).

Reference-assisted analysis We then evaluate the usefulness of reference snapshots with a controlled experiment on the ongoing leaking component ServiceS, which has

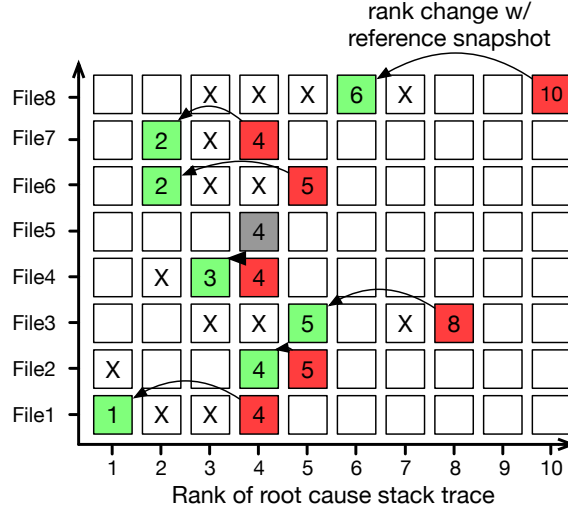


Figure 5.17: Ranks of root cause stack trace in diagnosis analyses on 8 trace files.

the most noises among the three ongoing leak cases. We randomly sample eight hosts that have leaking patterns and collect snapshots until the leaking stack appears. We feed eight collected trace files to RESIN and compare the analysis results with and without reference snapshots. Figure 5.17 shows the result. Green colored cells use reference snapshots and red colored cells do not. The numbers on cells represent the ranks. “X” marks the stack trace that gets filtered with the reference snapshots. Without the reference snapshots, the root cause stack trace ranks below the top three in all traces. With the reference snapshots, in 7 out of 8 traces, the root cause rank improves. In four traces, the rank rises to the top three, which largely narrows down the code regions developers need to investigate.

5.4.8 Runtime Overhead

As a production service, RESIN should not impose significant overhead on the hosts. For the detection component, since RESIN leverages the kernel to collect performance counters infrequently and offloads the analyses remotely, the overhead is minimal. The main source of overhead is the heap snapshot trace collection. We use

HasOverlap	Sessions	Nodes	25%	50%	75%	90%	95%	99%
FALSE	102,627	315	28	49	94	164	202	869
TRUE	165	31	38	50	59	86	241	888

Table 5.3: VM deployment time (seconds) impact by trace collection.

the VM deployment performance to quantify the end-to-end cost of trace collection, because VM deployment is the most important event for hosts and involves nearly all host services and triggers many critical code paths. A large overhead will be reflected in long deployment time.

We first check how many hosts RESIN performs trace collection on in November 2021. The result shows only 346 hosts are collected at least once, which is less than 0.1% of all nodes in a cluster. We then collect start and end timestamps of all VM deployment sessions and the heap snapshot tracing requests. We compare the timestamps in the two sets of events. In 315 (91%) of the 346 hosts, the deployment sessions do not have any overlap with tracing sessions, thus the tracing has no impact on these sessions.

Table 5.3 shows the end-to-end latency of the overlapped sessions compared to non-overlapped sessions: by 1 s for the median, and by 10 s for the 25th percentile. The latency increase could be notable for some short-duration deployments. However, this impact is limited to only a few sessions (0.16%) from a relatively small number of host nodes.

To measure the impact on memory size and CPU, we conduct experiments on two hosts that have active workloads. We trace one of the critical host processes. Figure 5.18 shows the memory and CPU usage during the experiment. Enabling tracing both slightly increases the average memory usage, 0.25 MB for host A and 0.53 MB for host B, and the CPU usage, 0.23% for host A and 0.22% for host B.

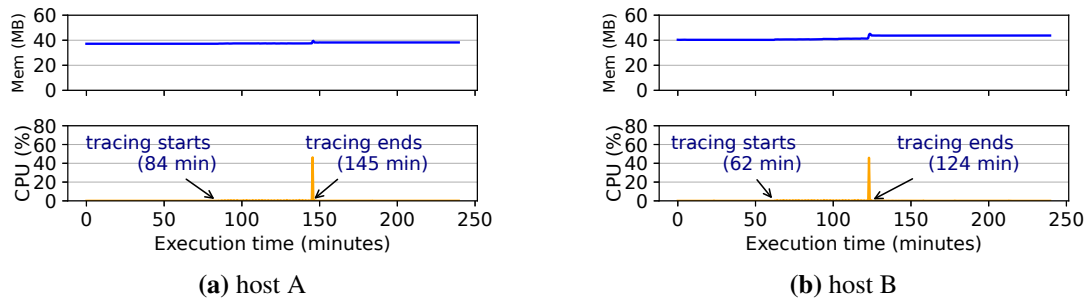


Figure 5.18: Memory size and CPU usage changes through tracing.

When doing snapshot and dumping the trace, there is a clear spike for CPU usage: around 46% in both hosts. The memory usage also increases, but not significantly: 1.93 MB for host A and 3.89 MB for host B. After the tracing, the memory usage remains at a slightly increased level due to a one-time initialization required by tracer. Overall, the CPU and memory usage costs are acceptable in production deployment.

5.4.9 Tuning Effort

The software components and workloads in Azure infrastructure undergo frequent changes. As part of RESIN’s design goals of minimizing false positives and false negatives, we aim to build robust algorithms that avoid fragile parameter tuning. For the detection part, throughout RESIN’s production operation, we only made one major parameter change. We updated the alert threshold for the final result $\langle \text{BucketSize}, \text{SeverityScore} \rangle$ from $\langle 50 \text{ MB}, 10 \rangle$ to $\langle 200 \text{ MB}, 40 \rangle$ around 3 months after deploying RESIN. There has not been further tuning since then. For the diagnosis part, except for the initial trial runs in which we were experimenting with the snapshot algorithms, the parameters for the completion triggers have not been tuned after the diagnosis engine was enabled in production.

5.5 Lessons and Limitations

Lessons While memory leaks are generally taken seriously, developers tend to postpone the investigation if there are no convincing hints. Presenting clear evidence in results is critical, and significantly improves developers' responsiveness.

Many teams write extensive test cases that check if allocations are freed. Some teams also implement their own version of memory leak detection tool in their testing cluster. Developers mentioned a major pain point is that the testing environment has significant discrepancies with the production environment. For example, in one case, developers mentioned *"We don't have an environment where ServiceH runs for a really long time with hosts undergoing reboots."*, otherwise, their testing would have caught the memory usage anomaly.

Our initial thought in designing the diagnosis module is to analyze the source code of the detected leaking component. We later found that finding the root cause stack traces is usually good enough for developers to debug the issue based on their own experience and domain knowledge.

For production services, safety is of high priority. The cloud infrastructure is a complex and dynamic environment. Some workflow in RESIN can be interrupted abruptly, *e.g.*, due to transient network issues, interference with other profiling tools. On one occasion, RESIN accidentally left the trace collection running and triggered alarms in the detection engine. We set three lines of protection to prevent similar issues: (i) limit collecting on same cluster within one hour five times at maximum to reduce side effects; (ii) a forced cleanup operation whether the profiling succeeds or not; (iii) a workflow that periodically checks logs and cleans up for runaway hosts.

Limitations The telemetry data RESIN analyzes is relatively coarse-grained. Even

the heap snapshot only contains limited information about allocations. Therefore, it has inherent inaccuracies and may miss detection of minor leak bugs. RESIN can be further enhanced by collecting more fine-grained signals, and leveraging semantic information from source code.

Developers may need to reproduce a reported memory leak issue for investigation or confirming bug fixes. But this is often challenging, because the issues are often triggered by complex workloads and rare conditions. RESIN does not address this challenge. We plan to automatically capture production triggering workloads for developers to reproduce leaks.

The patterns used in our heap snapshot trigger are based on empirical observations, which may be incomplete. Our classification method is simple. They can be improved with more comprehensive case studies and more advanced methods.

5.6 Conclusion

This chapter presents RESIN, an end-to-end service designed to tackle memory leaks in production cloud infrastructure. RESIN takes a divide-and-conquer approach to decompose the memory leak problem, and designs a multi-level solution with novel algorithms including bucketization-based pivot analysis, live heap snapshot strategy, and diagnosis analysis. RESIN has been running in Azure for more than 3 years, and successfully reduces low-memory-induced VM reboots and new VM allocation errors by $41\times$ and $10\times$, respectively.

Chapter 6

Related Work

6.1 Distributed Systems Failure Study

Understanding failures has been an important theme in distributed system literature, with a series of empirical studies [17, 176, 60, 84, 121, 82, 120, 7, 5, 151]. Arpaci-Dusseau and Arpaci-Dusseau propose the fail-stutter fault model [17]. Prabhakaran *et al.* analyze the fail-partial model for disks [183]. Correia *et al.* propose the ASC fault model [55]. Huang *et al.* propose a definition for gray failure in cloud [121]. Other notable studies include an examination of fail-slow performance faults in hardware by Gunawi *et al.* [82], an analysis of network partition-induced failures in cloud systems by Alquraan *et al.* [7], and a study on metastable failures leading to system overloading by Huang *et al.* [119].

The research in this dissertation contributes to this existing body of work in two ways: firstly, although previous studies have discussed certain types of failures such as partial failures in hardware, our research specifically targets failures in modern cloud software. We bring a unique perspective by examining runtime observations, and we broaden the scope to cover a wider array of root causes that extend beyond

hardware or network issues. Secondly, while traditionally studied failures often exhibit noticeable error signals like timeouts, our work fills a gap by concentrating on the less explored domain of silent semantic failures in distributed systems.

6.2 Failure Detection

Failure detection has been extensively studied in existing literature [42, 185, 3, 56, 45, 63, 88, 89, 141, 142, 143]. They primarily focus on detecting fail-stop failures in distributed systems. Chandra and Toueg [42] discuss several classes of unreliable crash failure detectors for solving consensus in asynchronous systems. GossipFD [185] proposes random gossiping for scalable crash failure detection; FALCON [141] constructs spies to achieve accurate crash detection. Failures we aim to address in this dissertation are beyond the scope of these detectors. Panorama [120] proposes to leverage observability in a system to detect gray failures [121]. 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 [129]. Instead our tool OmegaGen focuses on detecting and localizing partial failures within a process.

6.3 Failure Diagnosis

The field of distributed system failure diagnosis has made significant strides recently, as evidenced by numerous studies [218, 217, 189, 223, 26, 224]. Our approach shares a common aim with these work - the localization of the failure root cause. However, our focus and challenges are distinct. Whereas failure diagnosis research

is oriented towards assisting developers in understanding failure manifestations by reconstructing failure executions, our approach is primarily geared towards pinpointing the root cause to facilitate recovery. Furthermore, given that our tool is utilized in a production environment, we must address unique challenges such as ensuring the production system correctness and managing the performance penalty at runtime. Such challenges are generally not prominent in conventional failure diagnosis work.

6.4 Runtime Verification

Prior works have explored runtime assertion approach to verify distributed protocols [148, 147], file systems [66], and network functions [216]. Runtime verification [93] is also studied in embedded systems and Java benchmark programs [44]. Lu *et al.* propose a runtime checker for consistency violations [155]. Overall, there is a lack of runtime verification solutions for monitoring the semantic correctness of large-scale distributed system implementations. Our proposed tool Oathkeeper explores automatically extracting semantic rules to check a variety of semantics for large distributed systems.

One direction is to automatically infer likely invariants from software execution traces, *e.g.*, Daikon [61] and DIDUCE [90]. These works mainly focus on mining invariants on the relationship of program variables for single-component software, *e.g.*, `off < array.length`. These invariants are too low-level to capture the semantics of distributed systems. Dinv [80] is proposed to infer protocol invariants of program variables across nodes. It runs complex program slicing to instrument program variables influenced by network communication. It then uses Daikon to infer invariants from the logs of running the system’s test suite. I4 [158] infers inductive invariants

for verifying distributed protocols. Our tool, Oathkeeper, complements these works, but with a different emphasis. Instead of focusing on protocols and variable relations, we aim to infer high-level semantic rules for large distributed systems, the majority of which are not protocol-based. In contrast to Dinv, Oathkeeper does not depend on complex static analysis and, as a result, avoids associated inaccuracies and scalability issues. Uniquely, Oathkeeper utilizes past failures and semantic templates to extract semantic rules, carving its own niche in this field of study.

6.5 Distributed Tracing and Monitoring

Previous work has improved distributed system observability with a tracing approach [23, 64, 190, 159, 131]. Some solutions apply statistical techniques to collected metrics [27, 46, 51, 50, 191, 192, 127]. They are primarily designed to help diagnose performance issues in large and complex distributed systems. While they can help in pinpointing bottlenecks or inefficiencies, they target at a different problem scope and lack a high-level understanding of what these operations mean in the context of the overall system. They also cannot detect silent semantic failures since such failures do not cause obvious performance issues or error messages.

Chapter 7

Conclusion

Traditional system design methodologies often mask underlying failures to sustain high availability, operating under the assumption that entire components will fail, enabling backup components to detect these failures. However, the rise in complex system failures challenges this premise. This dissertation pivots towards managing complex failures by enhancing system observability, rather than following traditional fault-tolerance approaches by masking issues, to more effectively address them.

7.1 Limitations of Our Approach

While the evaluation results underscore the effectiveness of our solutions, our approach has a few limitations that we aim to address in the future work.

The generated checkers lack assurances of accuracy or completeness. Similar to criticisms commonly received by bug finding tools, our heuristic approach does not offer formal guarantees regarding the accuracy or thoroughness of the checkers, and they cannot unequivocally confirm or dismiss potential failures. These heuristics, grounded in empirical study observations, may not suit all system types. For

example, understanding failure characteristics in different cloud system paradigms, such as microservices and serverless computing, requires new studies.

The quality of checkers relies on the input sources. Automated systems can be convenient, but the quality of the produced checker depends on the quality of the program, tests, and traces. In our Oathkeeper project, we noticed that ad hoc test cases written by developers sometimes led to issues. Therefore, promoting standardized, rigorous test writing practices could be beneficial.

The potential safety risks are not fully mitigated. Our intrusive methodology might make developers wary of using our tools on production systems due to concerns about the safety of the generated code. Despite our tool providing transparency, such as OmegaGen displaying decompiled generated watchdogs, developers still yearn for stronger safety guarantees.

7.2 Lessons for Robust Cloud System Designs

Throughout our research, we have deepened our understanding of complex failures and distilled principles that could inform future robust cloud system designs. We discussed these principles in this section.

Addressing failures holistically. Complex failures often occur due to oversight of broader system impacts of local faults. Modern cloud systems, characterized by interactions between diverse modules, require a comprehensive perspective to detect both immediate and latent fault effects for effective error management. Developers must consider new approaches to integrate the system context when designing systems to

tolerate hypothetical failures.

Strengthening the system’s weakest link. The robustness of a system, much like a chain, is determined by its weakest link. While previous work predominantly focus on the central components of distributed systems such as consensus algorithms and transaction processing, we found many issues stem from auxiliary components. Interestingly, minor malfunctions, such as stuck issues during routine log dumping, can cause major disruptions like data loss. Therefore, developers should apply rigorous designs and thorough testing protocols for these components.

Constructing the system as a unified whole. Our work with the Oathkeeper project highlighted that current testing often targets specific functions and bugs, necessitating a more general approach. A focus on semantic properties and tighter integration with main programs should not only be applied to improve testing but also enhance ecosystem components like client interfaces and administrative tools.

7.3 Future Directions

Advancing mitigation and recovery techniques for complex failures. This dissertation mainly focuses on timely detection of complex failures in large distributed systems, meanwhile the subsequent question *how should systems appropriately react to complex failures upon detection* has yet to be fully explored. Current efforts prioritize mitigation and recovery to minimize the effects of failures. Yet, determining the right response proves to be a challenging task. Real-world production systems are still grappling with several challenges: (1) imprecise failure impact estimation

due to the complexity of interactions and internals in cloud systems; (2) misbehavior and side effects in the recovery process, which may paradoxically exacerbate the situation; and (3) high cost of existing recovery approaches which extends system downtime. Addressing these challenges presents valuable opportunities for future research.

A pragmatic verification framework for production systems. The primary focus of this dissertation has been on checking implementation-level codes. This objective aligns with, and supplements, existing endeavors such as the verification of distributed protocols. However, the task of verifying the implementation of distributed systems remains challenging. Formal methods often demand the formulation of a precise specification, a process that requires significant expertise and considerable effort, thus limiting its utility in real-world production environments. Through our work with Oathkeeper, we have discovered the potential to automatically derive system rules from test executions. It poses an interesting question: can we employ these inferred rules and harness model checking methods to verify correctness? This could mark a significant step towards making the process of system verification more efficient and practical.

Enhancing trust and privacy through system execution audits. OmegaGen has demonstrated the feasibility of using program analysis to automatically generate runtime checkers to improve reliability. We posit that the similar methodology could be expanded to enhance security and privacy in cloud systems. Nowadays, user focus on data utilization by tech companies has intensified, making it crucial to verify if these companies are adhering to their commitments [79]. Existing approaches assume the

platform is untrusted and interpret the program execution on the cloud as a black box. Instead of treating entities as hypothetical adversaries, it would be interesting to explore a new direction that involves cloud vendors in the solution. One conceivable avenue is to assist these vendors in creating auditing libraries for their services. These libraries could then provide verifiable proof that the system execution aligns with the declared properties. This proactive approach could, in turn, build more trust with users and enhance the overall data privacy landscape.

7.4 Concluding Remarks

Cloud systems, with their intricate architectures, present unique and challenging failure modes due to their extraordinary complexity. Through detailed studies and innovative strategies, we've proven our methods' effectiveness in improving these systems' availability. We aim for our work to not only solve current distributed system and cloud infrastructure issues but also inspire future designs to manage escalating complexity. Ultimately, our goal is to contribute to the creation of more dependable and resilient computing environments.

Bibliography

- [1] Andrei Agapi et al. “Routers for the cloud: Can the internet achieve 5-nines availability?” In: *IEEE Internet Computing* 15.5 (2011).
- [2] Marcos K Aguilera, Gérard Le Lann, and Sam Toueg. “On the impact of fast failure detectors on real-time fault-tolerant systems”. In: *International Symposium on Distributed Computing*. Springer. 2002, pp. 354–369.
- [3] Marcos K. Aguilera and Michael Walfish. “No Time for Asynchrony”. In: *Proceedings of the 12th Conference on Hot Topics in Operating Systems*. HotOS ’09. Monte Verità, Switzerland: USENIX Association, 2009, pp. 3–3. URL: <http://dl.acm.org/citation.cfm?id=1855568.1855571>.
- [4] Dr. Anamika Ahirwar and Anshu Parihar. “IMPACT OF CLOUD COMPUTING ON BUSINESS”. In: *BSSS journal of computer* 12 (2021), pp. 90–97. DOI: 10.51767/jc1210.
- [5] Mohammed Alfatafta et al. “Toward a Generic Fault Tolerance Technique for Partial Network Partitioning”. In: *14th USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’20. USENIX Association, 2020, pp. 351–368. ISBN: 978-1-939133-19-9. URL: <https://www.usenix.org/conference/osdi20/presentation/alfatafta>.
- [6] *Alibaba Cloud Reports IO Hang Error in North China*. <https://equalocean.com/technology/20190303-alibaba-cloud-reports-io-hang-error-in-north-china>.
- [7] Ahmed Alquraan et al. “An Analysis of Network-Partitioning Failures in Cloud Systems”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI ’18. Carlsbad, CA, USA: USENIX Association, 2018, 51–68. ISBN: 9781931971478.
- [8] Peter A Alsberg and John D Day. “A principle for resilient sharing of distributed resources”. In: *Proceedings of the 2nd international conference on Software engineering*. 1976, pp. 562–570.

- [9] Amazon. *AWS service outage on October 22nd, 2012*. <https://aws.amazon.com/message/680342>.
- [10] *Amazon Cloud Outage Hits Customers Including Roku, Adobe*. <https://www.bloomberg.com/news/articles/2020-11-25/amazon-web-services-outage-hits-cloud-customers/>.
- [11] *Amazon EC2 Service Level Agreement*. <https://aws.amazon.com/ec2/sla/>.
- [12] *Amazon 'missed out on \$34m in sales during internet outage'*. <https://www.independent.co.uk/news/business/amazon-down-internet-outage-sales-b1861737.html>.
- [13] *Apache Cassandra: Some useful JMX metrics to monitor*. <https://medium.com/@foundev/apache-cassandra-some-useful-jmx-metrics-to-monitor-7f1d3ede294a>.
- [14] *Apache Module mod_proxy_hcheck*. https://httpd.apache.org/docs/2.4/mod/mod_proxy_hcheck.html.
- [15] *Apache ZooKeeper Releases*. <https://zookeeper.apache.org/releases.html>.
- [16] Joe Armstrong. "Making reliable distributed systems in the presence of software errors". PhD thesis. The Royal Institute of Technology, Stockholm, Sweden, 2003.
- [17] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. "Fail-Stutter Fault Tolerance". In: *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. HotOS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 33–. URL: <http://dl.acm.org/citation.cfm?id=874075.876394>.
- [18] Mona Attariyan, Michael Chow, and Jason Flinn. "X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI '12. Hollywood, CA, USA, 2012, pp. 307–320.
- [19] Mona Attariyan and Jason Flinn. "Automating Configuration Troubleshooting with Dynamic Information Flow Analysis". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI '10. Vancouver, BC, Canada, 2010, pp. 1–11.
- [20] *AWS Post-Event Summaries*. <https://aws.amazon.com/premiumsupport/technology/pes/>.

- [21] *AWS US East region endures eight-hour wobble thanks to 'Stuck IO' in Elastic Block Store*. https://www.theregister.com/2021/09/28/aws_east_brownout/.
- [22] *Axios. Amazon Web Services investing \$7.8 billion in new data centers*. <https://www.axios.com/local/columbus/2023/06/27/amazon-data-center-investment-ohio-2023>. 2023.
- [23] Paul Barham et al. "Using Magpie for Request Extraction and Workload Modelling". In: *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. San Francisco, CA: USENIX Association, 2004. URL: <https://www.usenix.org/conference/osdi-04/using-magpie-request-extraction-and-workload-modelling>.
- [24] Salman A Baset. "Cloud SLAs: present and future". In: *ACM SIGOPS Operating Systems Review* 46.2 (2012), pp. 57–66.
- [25] Marin Bertier, Olivier Marin, and Pierre Sens. "Implementation and performance evaluation of an adaptable failure detector". In: *Proceedings International Conference on Dependable Systems and Networks*. IEEE. 2002, pp. 354–363.
- [26] Ranjita Bhagwan et al. "Orca: Differential Bug Localization in {Large-Scale} Services". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 493–509.
- [27] Peter Bodik et al. "Fingerprinting the datacenter: automated classification of performance crises". In: *Proceedings of the 5th European conference on Computer systems*. 2010, pp. 111–124.
- [28] Michael D. Bond and Kathryn S. McKinley. "Bell: Bit-Encoding Online Memory Leak Detection". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '06. San Jose, California, USA: Association for Computing Machinery, 2006, 61–72. ISBN: 1595934510. DOI: 10.1145/1168857.1168866. URL: <https://doi.org/10.1145/1168857.1168866>.
- [29] Mike Burrows. "The Chubby Lock Service for Loosely-coupled Distributed Systems". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 335–350. ISBN: 1-931971-47-1. URL: <http://dl.acm.org/citation.cfm?id=1298455.1298487>.

- [30] Brad Calder et al. “Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal: ACM, 2011, pp. 143–157. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043571. URL: <http://doi.acm.org/10.1145/2043556.2043571>.
- [31] George Candea et al. “Microreboot — A Technique for Cheap Recovery”. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*. OSDI ’04. San Francisco, CA: USENIX Association, 2004, pp. 31–44. URL: <http://dl.acm.org/citation.cfm?id=1251254.1251257>.
- [32] *Cassandra-10477: java.lang.AssertionError in StorageProxy.submitHint*. <https://issues.apache.org/jira/browse/CASSANDRA-10477>.
- [33] *Cassandra-5229: streaming tasks hang in netstats*. <https://issues.apache.org/jira/browse/CASSANDRA-5229>.
- [34] *Cassandra-6364: Commit log executor dies and causes unflushed writes to quickly accumulate*. <https://issues.apache.org/jira/browse/CASSANDRA-6364>.
- [35] *Cassandra-6415: Snapshot repair blocks forever if something happens to the remote response*. <https://issues.apache.org/jira/browse/CASSANDRA-6415>.
- [36] *Cassandra-6788: Race condition silently kills thrift server*. <https://issues.apache.org/jira/browse/CASSANDRA-6788>.
- [37] *Cassandra-8447: Nodes stuck in CMS GC cycle with very little traffic when compaction is enabled*. <https://issues.apache.org/jira/browse/CASSANDRA-8447>.
- [38] *Cassandra-9486: LazilyCompactedRow accumulates all expired RangeTombstones*. <https://issues.apache.org/jira/browse/CASSANDRA-9486>.
- [39] *Cassandra-9549: Memory leak in Ref.GlobalState due to pathological ConcurrentLinkedQueue.remove behaviour*. <https://issues.apache.org/jira/browse/CASSANDRA-9549>.
- [40] *Cassandra: demystify failure detector, consider partial failure handling, latency optimizations*. <https://issues.apache.org/jira/browse/CASSANDRA-3927>.
- [41] Miguel Castro, Barbara Liskov, et al. “Practical byzantine fault tolerance”. In: *OsDI*. Vol. 99. 1999. 1999, pp. 173–186.

- [42] Tushar Deepak Chandra and Sam Toueg. “Unreliable Failure Detectors for Reliable Distributed Systems”. In: *J. ACM* 43.2 (1996), pp. 225–267. ISSN: 0004-5411. DOI: 10.1145/226643.226647. URL: <http://doi.acm.org/10.1145/226643.226647>.
- [43] K. Mani Chandy and Leslie Lamport. “Distributed Snapshots: Determining Global States of Distributed Systems”. In: *ACM Trans. Comput. Syst.* 3.1 (1985), 63–75. ISSN: 0734-2071. DOI: 10.1145/214451.214456. URL: <https://doi.org/10.1145/214451.214456>.
- [44] Feng Chen and Grigore Roşu. “Mop: An Efficient and Generic Runtime Verification Framework”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*. OOPSLA ’07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, 569–588. ISBN: 9781595937865. DOI: 10.1145/1297027.1297069. URL: <https://doi.org/10.1145/1297027.1297069>.
- [45] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. “On the Quality of Service of Failure Detectors”. In: *IEEE Trans. Comput.* 51.5 (2002), pp. 561–580. ISSN: 0018-9340. DOI: 10.1109/TC.2002.1004595. URL: <https://doi.org/10.1109/TC.2002.1004595>.
- [46] Michael Chow et al. “The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, 2014, pp. 217–231. ISBN: 978-1-931971-16-4. URL: <http://dl.acm.org/citation.cfm?id=2685048.2685066>.
- [47] Cisco. *Cisco Annual Internet Report*. <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>. 2020.
- [48] Christopher Clark et al. “Live Migration of Virtual Machines”. In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*. NSDI ’05. USA: USENIX Association, 2005, 273–286.
- [49] Allen Clement et al. “UpRight Cluster Services”. In: *Proc. ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, 2009, pp. 277–290. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629602. URL: <http://doi.acm.org/10.1145/1629575.1629602>.

- [50] Ira Cohen et al. “Capturing, Indexing, Clustering, and Retrieving System History”. In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. Brighton, United Kingdom, 2005, pp. 105–118. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095821. URL: <http://doi.acm.org/10.1145/1095810.1095821>.
- [51] Ira Cohen et al. “Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control”. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*. San Francisco, CA, 2004, pp. 16–16. URL: <http://dl.acm.org/citation.cfm?id=1251254.1251270>.
- [52] *Consul health check*. <https://www.consul.io/docs/agent/checks.html>.
- [53] Brian F. Cooper et al. “Benchmarking Cloud Serving Systems with YCSB”. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC ’10. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152. URL: <http://doi.acm.org/10.1145/1807128.1807152>.
- [54] James C. Corbett et al. “Spanner: Google’s Globally-distributed Database”. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. Hollywood, CA, USA: USENIX Association, 2012, pp. 251–264. ISBN: 978-1-931971-96-6. URL: <http://dl.acm.org/citation.cfm?id=2387880.2387905>.
- [55] Miguel Correia et al. “Practical Hardening of Crash-tolerant Systems”. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC’12. Boston, MA: USENIX Association, 2012, pp. 41–41. URL: <http://dl.acm.org/citation.cfm?id=2342821.2342862>.
- [56] Abhinandan Das, Indranil Gupta, and Ashish 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. Washington, DC, USA: IEEE Computer Society, 2002, pp. 303–312. ISBN: 0-7695-1597-5. URL: <http://dl.acm.org/citation.cfm?id=647883.738420>.
- [57] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI’04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: <http://dl.acm.org/citation.cfm?id=1251254.1251264>.

- [58] Giuseppe DeCandia et al. “Dynamo: Amazon’s Highly Available Key-value Store”. In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP ’07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: <http://doi.acm.org/10.1145/1294261.1294281>.
- [59] *Distributed database benchmark tester*. <https://github.com/etcd-io/dbtester>.
- [60] Thanh Do et al. “Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: ACM, 2013, 14:1–14:14. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523627. URL: <http://doi.acm.org/10.1145/2523616.2523627>.
- [61] Michael D. Ernst et al. “The Daikon System for Dynamic Detection of Likely Invariants”. In: *Sci. Comput. Program.* 69.1–3 (2007), 35–45. ISSN: 0167-6423. DOI: 10.1016/j.scico.2007.01.015. URL: <https://doi.org/10.1016/j.scico.2007.01.015>.
- [62] Gang Fan et al. “Smoke: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code”. In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE ’19. Montreal, Quebec, Canada: IEEE Press, 2019, 72–82. DOI: 10.1109/ICSE.2019.00025. URL: <https://doi.org/10.1109/ICSE.2019.00025>.
- [63] Christof Fetzer. “Perfect Failure Detection in Timed Asynchronous Systems”. In: *IEEE Trans. Comput.* 52.2 (2003), pp. 99–112. ISSN: 0018-9340. DOI: 10.1109/TC.2003.1176979. URL: <http://dx.doi.org/10.1109/TC.2003.1176979>.
- [64] Rodrigo Fonseca et al. “X-Trace: A Pervasive Network Tracing Framework”. In: *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. Cambridge, MA: USENIX Association, 2007. URL: <https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework>.
- [65] Apache Software Foundation. *HDFS High Availability Using the Quorum Journal Manager*. <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html>.
- [66] Daniel Fryer et al. “Recon: Verifying File System Consistency at Runtime”. In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. FAST ’12. San Jose, CA: USENIX Association, 2012, pp. 7–7. URL: <http://dl.acm.org/citation.cfm?id=2208461.2208468>.

- [67] Mohammadreza Ghanavati et al. “Memory and resource leak defects and their repairs in Java projects”. In: *Empirical Software Engineering* 25.1 (2020), pp. 678–718. DOI: 10.1007/s10664-019-09731-8. URL: <https://doi.org/10.1007/s10664-019-09731-8>.
- [68] Mohammadreza Ghanavati et al. “Memory and Resource Leak Defects in Java Projects: An Empirical Study”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 410–411. ISBN: 9781450356633. DOI: 10.1145/3183440.3195032. URL: <https://doi.org/10.1145/3183440.3195032>.
- [69] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: ACM, 2003, pp. 29–43. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945450. URL: <http://doi.acm.org/10.1145/945445.945450>.
- [70] Evan Gilman. *The discovery of Apache ZooKeeper’s poison packet*. <https://www.pagerduty.com/blog/the-discovery-of-apache-zookeepers-poison-packet>. 2015.
- [71] *GoCardless service outage on October 10th, 2017*. <https://gocardless.com/blog/incident-review-api-and-dashboard-outage-on-10th-october>.
- [72] Joseph A. Goguen. “Semantics of Computation”. In: *Proceedings of the First International Symposium on Category Theory Applied to Computation and Control*. Berlin, Heidelberg: Springer-Verlag, 1974, 151–163. ISBN: 3540071423.
- [73] *Gone in Minutes, Out for Hours: Outage Shakes Facebook*. <https://www.nytimes.com/2021/10/04/technology/facebook-down.html>.
- [74] Google. *Twilio Billing Incident Post-Mortem: Breakdown, Analysis and Root Cause*. <https://www.twilio.com/blog/2013/07/billing-incident-post-mortem-breakdown-analysis-and-root-cause.html>.
- [75] *Google Cloud Infrastructure Incident #20013*. <https://status.cloud.google.com/incident/zall/20013>.
- [76] *Google Cloud Service Health - Incidents*. <https://status.cloud.google.com/summary>.
- [77] *Google Cloud Storage Incident #17005*. <https://status.cloud.google.com/incident/storage/17005>.

- [78] *Google Compute Engine Incident 17008*. <https://status.cloud.google.com/incident/compute/17008>. 2017.
- [79] *Google Creating trust through transparency*. <https://cloud.google.com/security/transparency>.
- [80] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. “Inferring and Asserting Distributed System Invariants”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 1149–1159. ISBN: 9781450356381. DOI: 10.1145/3180155.3180199. URL: <https://doi.org/10.1145/3180155.3180199>.
- [81] Haryadi S. Gunawi et al. “EIO: Error Handling is Occasionally Correct”. In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. FAST ’08. San Jose, California: USENIX Association, 2008, 14:1–14:16. URL: <http://dl.acm.org/citation.cfm?id=1364813.1364827>.
- [82] Haryadi S. Gunawi et al. “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. Oakland, CA, USA: USENIX Association, 2018, pp. 1–14. ISBN: 978-1-931971-42-3. URL: <http://dl.acm.org/citation.cfm?id=3189759.3189761>.
- [83] Haryadi S Gunawi et al. “What bugs live in the cloud? a study of 3000+ issues in cloud systems”. In: *Proceedings of the ACM symposium on cloud computing*. 2014, pp. 1–14.
- [84] Haryadi S. Gunawi et al. “Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages”. In: *Proceedings of the 7th ACM Symposium on Cloud Computing*. SOCC ’16. Santa Clara, CA, USA, 2016, pp. 1–16. ISBN: 978-1-4503-4525-5. DOI: 10.1145/2987550.2987583. URL: <http://doi.acm.org/10.1145/2987550.2987583>.
- [85] Chuanxiong Guo et al. “Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis”. In: *Proceedings of the 2015 ACM SIGCOMM Conference*. SIGCOMM ’15. London, United Kingdom: ACM, 2015, pp. 139–152. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787496. URL: <http://doi.acm.org/10.1145/2785956.2787496>.
- [86] Ashish Gupta and Jeff 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.

- [87] Brian Hackett and Radu Rugina. “Region-Based Shape Analysis with Tracked Locations”. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: Association for Computing Machinery, 2005, 310–323. ISBN: 158113830X. DOI: 10.1145/1040305.1040331. URL: <https://doi.org/10.1145/1040305.1040331>.
- [88] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. “PeerReview: Practical Accountability for Distributed Systems”. In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP ’07. Stevenson, Washington, USA: ACM, 2007, pp. 175–188. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294279. URL: <http://doi.acm.org/10.1145/1294261.1294279>.
- [89] Andreas Haeberlen and Petr Kuznetsov. “The Fault Detection Problem”. In: *Proceedings of the 13th International Conference on Principles of Distributed Systems*. OPODIS ’09. Nîmes, France: Springer-Verlag, 2009, pp. 99–114. ISBN: 978-3-642-10876-1. DOI: 10.1007/978-3-642-10877-8_10. URL: http://dx.doi.org/10.1007/978-3-642-10877-8_10.
- [90] Sudheendra Hangal and Monica S. Lam. “Tracking down Software Bugs Using Automatic Anomaly Detection”. In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE ’02. Orlando, Florida: Association for Computing Machinery, 2002, 291–301. ISBN: 158113472X. DOI: 10.1145/581339.581377. URL: <https://doi.org/10.1145/581339.581377>.
- [91] Reed Hastings and Bob Joyce. “Purify: Fast detection of memory leaks and access errors”. In: *Proc. of the Winter 1992 USENIX Conference*. Berkeley, 1992, 125–138.
- [92] Matthias Hauswirth and Trishul M. Chilimbi. “Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling”. In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’04. Boston, MA, USA: Association for Computing Machinery, 2004, 156–164. ISBN: 1581138040. DOI: 10.1145/1024393.1024412. URL: <https://doi.org/10.1145/1024393.1024412>.
- [93] Klaus Havelund and Grigore Roşu. “Runtime Verification”. In: *Computer Aided Verification (CAV ’01) satellite workshop (ENTCS) 55* (2001).

- [94] Naohiro Hayashibara et al. “The ϕ Accrual Failure Detector”. In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*. SRDS '04. Florianópolis, Brazil: IEEE Computer Society, 2004, pp. 66–78. ISBN: 0-7695-2239-4. URL: <http://dl.acm.org/citation.cfm?id=1032662.1034350>.
- [95] *HBASE-11536: Puts of region location to Meta may be out of order which causes inconsistent of region location*. <https://issues.apache.org/jira/browse/HBASE-11536>.
- [96] *HBASE-16081: Removing peer in replication not gracefully finishing blocks WAL rolling*. <https://issues.apache.org/jira/browse/HBASE-16081>.
- [97] *HBASE-16429: FSHLog deadlock if rollWriter called when ring buffer filled with appends*. <https://issues.apache.org/jira/browse/HBASE-16429>.
- [98] *HBase-17125: Inconsistent result when use filter to read data*. <https://issues.apache.org/jira/browse/HBASE-17125>.
- [99] *HBASE-18137: Empty WALs cause replication queue to get stuck*. <https://issues.apache.org/jira/browse/HBASE-18137>.
- [100] *HBASE-21357: Reader thread encounters out of memory error*. <https://issues.apache.org/jira/browse/HBASE-21357>.
- [101] *HBASE-21464: Splitting blocked with meta NSRE during split transaction*. <https://issues.apache.org/jira/browse/HBASE-21464>.
- [102] *HDFS-11352: Potential deadlock in NN when failing over*. <https://issues.apache.org/jira/browse/HDFS-11352>.
- [103] *HDFS-11377: Balancer hung due to no available mover threads*. <https://issues.apache.org/jira/browse/HDFS-11377>.
- [104] *HDFS-12070: Failed block recovery leaves files open indefinitely and at risk for data loss*. <https://issues.apache.org/jira/browse/HDFS-12070>.
- [105] *HDFS-12217: HDFS snapshots doesn't capture all open files when one of the open files is deleted*. <https://issues.apache.org/jira/browse/HDFS-12217>.
- [106] *HDFS-14359: Inherited ACL permissions masked when parent directory does not exist (mkdir -p)*. <https://issues.apache.org/jira/browse/HDFS-14359>.
- [107] *HDFS-2882: DN continues to start up, even if block pool fails to initialize*. <https://issues.apache.org/jira/browse/HDFS-2882>.
- [108] *HDFS-4176: EditLogTailer should call rollEdits with a timeout*. <https://issues.apache.org/jira/browse/HDFS-4176>.

- [109] *HDFS-4233: NN keeps serving even after no journals started while rolling edit*. <https://issues.apache.org/jira/browse/HDFS-4233>.
- [110] *HDFS-8429: Error in DomainSocketWatcher causes others threads to be stuck threads*. <https://issues.apache.org/jira/browse/HDFS-8429>.
- [111] *HDFS-9083: Replication violates block placement policy*. <https://issues.apache.org/jira/browse/HDFS-12070>.
- [112] *HDFS Short-Circuit Local Reads*. <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html>.
- [113] David L. Heine and Monica S. Lam. “A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI ’03. San Diego, California, USA: Association for Computing Machinery, 2003, 168–181. ISBN: 1581136625. DOI: 10.1145/781131.781150. URL: <https://doi.org/10.1145/781131.781150>.
- [114] Chi Ho et al. “Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures.” In: *NSDI*. Vol. 8. 2008, pp. 175–188.
- [115] 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. 2017, pp. 484–491. DOI: 10.1109/ICST.2017.55.
- [116] *How to Monitor Zookeeper*. <https://blog.serverdensity.com/how-to-monitor-zookeeper/>.
- [117] Yigong Hu, Gongqi Huang, and Peng Huang. “Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution”. In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. OSDI ’20. USENIX, 2020.
- [118] Lexiang Huang et al. “Metastable Failures in the Wild”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 73–90.
- [119] Lexiang Huang et al. “Metastable Failures in the Wild”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, 2022, pp. 73–90. ISBN: 978-1-939133-28-1. URL: <https://www.usenix.org/conference/osdi22/presentation/huang-lexiang>.

- [120] Peng Huang et al. “Capturing and Enhancing In Situ System Observability for Failure Detection”. In: *13th USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’18. Carlsbad, CA: USENIX Association, 2018, pp. 1–16. ISBN: 978-1-931971-47-8. URL: <https://www.usenix.org/conference/osdi18/presentation/huang>.
- [121] Peng Huang et al. “Gray Failure: The Achilles’ Heel of Cloud-Scale Systems”. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS ’17. British Columbia, Canada: ACM, 2017.
- [122] Wei Huang et al. “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. Tucson, Arizona, USA: ACM, 2012, pp. 879–896. ISBN: 978-1-4503-1561-6. DOI: 10.1145/2384616.2384680. URL: <http://doi.acm.org/10.1145/2384616.2384680>.
- [123] David P Jasper. “A discussion of checkpoint restart”. In: *Software Age* 3.10 (1969), pp. 9–14.
- [124] Mareike Jenner. *Netflix and the Re-invention of Television*. 2018. ISBN: 978-3-319-94315-2. DOI: 10.1007/978-3-319-94316-9.
- [125] Simon Holm Jensen et al. “MemInsight: Platform-Independent Memory Debugging for JavaScript”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE ’15. Bergamo, Italy: Association for Computing Machinery, 2015, 345–356. ISBN: 9781450336758. DOI: 10.1145/2786805.2786860. URL: <https://doi.org/10.1145/2786805.2786860>.
- [126] Guoliang Jin et al. “Understanding and detecting real-world performance bugs”. In: *ACM SIGPLAN Notices* 47.6 (2012), pp. 77–88.
- [127] Anshul Jindal et al. “Online Memory Leak Detection in the Cloud-based Infrastructures”. In: *International Conference on Service-Oriented Computing*. ICSOC ’20. Dubai, United Arab Emirates, 2020, pp. 188–200.
- [128] Changhee Jung et al. “Automated Memory Leak Detection for Production Use”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE ’14. Hyderabad, India: Association for Computing Machinery, 2014, 825–836. ISBN: 9781450327565. DOI: 10.1145/2568225.2568311. URL: <https://doi.org/10.1145/2568225.2568311>.
- [129] *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>.

- [130] *KAFKA-2960: DelayedProduce may cause message loss during repeated leader change*. <https://issues.apache.org/jira/browse/KAFKA-2960>.
- [131] Jonathan Kaldor et al. “Canopy: An End-to-End Performance Tracing And Analysis System”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, 34–50. ISBN: 9781450350853. DOI: 10 . 1145 / 3132747 . 3132749. URL: <https://doi.org/10.1145/3132747.3132749>.
- [132] Charles Killian et al. “Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code”. In: *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*. NSDI ’07. Cambridge, MA: USENIX Association, 2007, p. 18.
- [133] Dave King. *Partial Failures are Worse Than Total Failures*. <https://www.tildedave.com/2014/03/01/application-failure-scenarios-with-cassandra.html>. 2014.
- [134] Ramakrishna Kotla et al. “Zyzyva: speculative byzantine fault tolerance”. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 2007, pp. 45–58.
- [135] Kostas Kougiou. *Java cloning library*. <https://github.com/kostaskougiou/cloning>.
- [136] Leslie Lamport. “Paxos made simple”. In: *ACM Sigact News* 32.4 (2001), pp. 18–25.
- [137] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine generals problem”. In: *Concurrency: the works of leslie lamport*. 2019, pp. 203–226.
- [138] *LeakSanitizer – Clang 13 documentation*. <https://clang.llvm.org/docs/LeakSanitizer.html>.
- [139] Sangho Lee, Changhee Jung, and Santosh Pande. “Detecting Memory Leaks through Introspective Dynamic Behavior Modelling Using Machine Learning”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE ’14. Hyderabad, India: Association for Computing Machinery, 2014, 814–824. ISBN: 9781450327565. DOI: 10 . 1145 / 2568225 . 2568307. URL: <https://doi.org/10.1145/2568225.2568307>.
- [140] Tanakorn Leesatapornwongsa et al. “{SAMC}:{Semantic-Aware} Model Checking for Fast Discovery of Deep Bugs in Cloud Systems”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 399–414.

- [141] Joshua B. Leners et al. “Detecting Failures in Distributed Systems with the Falcon Spy Network”. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal, 2011, pp. 279–294.
- [142] Joshua B. Leners et al. “Improving Availability in Distributed Systems with Failure Informers”. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. NSDI ’13. Lombard, IL, 2013, pp. 427–442. URL: <http://dl.acm.org/citation.cfm?id=2482626.2482667>.
- [143] Joshua B. Leners et al. “Taming Uncertainty in Distributed Systems with Help from the Network”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: ACM, 2015, 9:1–9:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741976. URL: <http://doi.acm.org/10.1145/2741948.2741976>.
- [144] Sebastien Levy et al. “Predictive and Adaptive Failure Mitigation to Avert Production Cloud VM Interruptions”. In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’20. USENIX, 2020.
- [145] Guangpu Li et al. “Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, 162–180. ISBN: 9781450368735. DOI: 10.1145/3341301.3359638. URL: <https://doi.org/10.1145/3341301.3359638>.
- [146] Ze Li et al. “Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure”. In: *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. NSDI ’20. Santa Clara, CA: USENIX, 2020. URL: <https://www.usenix.org/conference/nsdi20/presentation/li>.
- [147] Xuezheng Liu et al. “D³S: Debugging Deployed Distributed Systems”. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI ’08. USA: USENIX Association, 2008, 423–437. ISBN: 1119995555221.
- [148] Xuezheng Liu et al. “WiDS Checker: Combating Bugs in Distributed Systems”. In: *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*. NSDI ’07. Cambridge, MA: USENIX Association, 2007.
- [149] *LMAX Disruptor*. <https://lmax-exchange.github.io/disruptor/>.

- [150] Chang Lou, Peng Huang, and Scott Smith. “Comprehensive and Efficient Runtime Checking in System Software through Watchdogs”. In: *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*. HotOS ’19. Bertinoro, Italy: ACM, 2019. ISBN: 978-1-4503-6727-1/19/05. DOI: 10.1145/3317550.3321440. URL: <https://doi.org/10.1145/3317550.3321440>.
- [151] Chang Lou, Peng Huang, and Scott Smith. “Understanding, Detecting and Localizing Partial Failures in Large System Software”. In: *17th USENIX Symposium on Networked Systems Design and Implementation*. NSDI ’20. Santa Clara, CA: USENIX Association, 2020, pp. 559–574. ISBN: 978-1-939133-13-7. URL: <https://www.usenix.org/conference/nsdi20/presentation/lou>.
- [152] Chang Lou, Yuzhuo Jing, and Peng Huang. *A Promise Is Not a Promise—Demystifying and Checking Silent Semantic Violations in Large Distributed Systems*. Tech. rep. Johns Hopkins University, 2022.
- [153] Chang Lou, Yuzhuo Jing, and Peng Huang. “Demystifying and Checking Silent Semantic Violations in Large Distributed Systems”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA, 2022, pp. 91–107.
- [154] Chang Lou et al. “RESIN: A Holistic Service for Dealing with Memory Leaks in Production Cloud Infrastructure”. In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’22. Carlsbad, CA, USA: USENIX Association, 2022, pp. 109–125. ISBN: 978-1-939133-28-1. URL: <https://www.usenix.org/conference/osdi22/presentation/lou-resin>.
- [155] Haonan Lu et al. “Existential Consistency: Measuring and Understanding Consistency at Facebook”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. Monterey, California: Association for Computing Machinery, 2015, 295–310. ISBN: 9781450338349. DOI: 10.1145/2815400.2815426. URL: <https://doi.org/10.1145/2815400.2815426>.
- [156] Jie Lu et al. “CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, 114–130. ISBN: 9781450368735. DOI: 10.1145/3341301.3359645. URL: <https://doi.org/10.1145/3341301.3359645>.

- [157] Shan Lu et al. “MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs”. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 2007, pp. 103–116.
- [158] Haojun Ma et al. “I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, 370–384. ISBN: 9781450368735. DOI: 10.1145/3341301.3359651. URL: <https://doi.org/10.1145/3341301.3359651>.
- [159] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016. URL: <https://www.usenix.org/conference/atc16/technical-sessions/presentation/mace>.
- [160] A. Mahmood and E. J. McCluskey. “Concurrent error detection using watchdog processors – a survey”. In: *IEEE Transactions on Computers* 37.2 (1988), pp. 160–174. ISSN: 0018-9340. DOI: 10.1109/12.2145.
- [161] *MapReduce-3634: Dispatchers in daemons get exceptions and silently stop processing*. <https://issues.apache.org/jira/browse/MAPREDUCE-3634>.
- [162] *MapReduce-6190: Job stuck for hours because one of the mappers never started up fully*. <https://issues.apache.org/jira/browse/MAPREDUCE-6190>.
- [163] *MapReduce-6351: Circular wait in handling errors causes reducer to hang in copy phase*. <https://issues.apache.org/jira/browse/MAPREDUCE-6351>.
- [164] *MapReduce-6957: Shuffle hangs after a node manager connection timeout*. <https://issues.apache.org/jira/browse/MAPREDUCE-6957>.
- [165] *Mesos-8830: Agent gc on old slave sandboxes could empty persistent volume data*. <https://issues.apache.org/jira/browse/MESOS-8830>.
- [166] Microsoft. *Microsoft Azure Storage Disruption in US South on December 28th, 2012*. <http://blogs.msdn.com/b/windowsazure/archive/2013/01/16/details-of-the-december-28th-2012-windows-azure-storage-disruption-in-us-south.aspx>.
- [167] Microsoft. *Office 365 Service Incident on November 13th, 2013*. <https://blogs.office.com/2012/11/13/update-on-recent-customer-issues/>.

- [168] Microsoft. *Windows kernel API: ExAllocatePoolWithTag*. <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-exallocatepoolwithtag>.
- [169] *Microsoft Azure Virtual Machines Service Level Agreement*. <https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/>.
- [170] *Microsoft Blames "Severe Weather" for Azure Cloud Outage*. <https://www.datacenterknowledge.com/uptime/microsoft-blames-severe-weather-azure-cloud-outage>.
- [171] *Microsoft's virtual datacenter grounds the cloud in reality*. <https://news.microsoft.com/source/features/innovation/microsofts-virtual-datacenter-grounds-the-cloud-in-reality/>.
- [172] *mod_proxy_ajp: mixed up response after client connection abort*. https://bz.apache.org/bugzilla/show_bug.cgi?id=53727.
- [173] *MongoDB-12355: add "invariant" for invariant checking in server code*. <https://jira.mongodb.org/browse/SERVER-12355>.
- [174] *MongoDB-50971: Invariant failure, WT_NOTFOUND: item not found*. <https://jira.mongodb.org/browse/SERVER-50971>.
- [175] Donny Nadolny. "Debugging Distributed Systems". In: *SREcon 2016*. Santa Clara, CA, 2016. URL: <https://www.usenix.org/node/195676>.
- [176] David Oppenheimer, Archana Ganapathi, and David A. Patterson. "Why Do Internet Services Fail, and What Can Be Done About It?" In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*. USITS '03. Seattle, WA, 2003. URL: <http://dl.acm.org/citation.cfm?id=1251460.1251461>.
- [177] Maksim Orlovich and Radu Rugina. "Memory Leak Analysis by Contradiction". In: *Proceedings of the 13th International Static Analysis Symposium*. SAS '06. Korea, 2006, pp. 405–424. DOI: 10.1007/11823230_26. URL: https://doi.org/10.1007/11823230_26.
- [178] *Overview of the JMX Technology*. <https://docs.oracle.com/javase/tutorial/jmx/overview/index.html>.
- [179] Biswaranjan Panda et al. "IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, 2019, pp. 47–62. ISBN: 978-1-939133-03-8. URL: <https://www.usenix.org/conference/atc19/presentation/panda>.

- [180] David Patterson et al. *Recovery Oriented Computing (ROC): Motivation, Definition, Techniques*, tech. rep. USA, 2002.
- [181] David Patterson et al. *Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. Tech. rep. UCB/CSD-02-1175. EECS Department, University of California, Berkeley, 2002. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2002/5574.html>.
- [182] David A. Patterson, Garth Gibson, and Randy H. Katz. “A Case for Redundant Arrays of Inexpensive Disks (RAID)”. In: *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. Chicago, Illinois, USA, 1988, pp. 109–116. ISBN: 0-89791-268-3. DOI: 10.1145/50202.50214. URL: <http://doi.acm.org/10.1145/50202.50214>.
- [183] Vijayan Prabhakaran et al. “IRON File Systems”. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP ’05. Brighton, United Kingdom: ACM, 2005, pp. 206–220. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095830. URL: <http://doi.acm.org/10.1145/1095810.1095830>.
- [184] Derek Rayside and Lucy Mendel. “Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks”. In: *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, 194–203. ISBN: 9781595938824. DOI: 10.1145/1321631.1321661. URL: <https://doi.org/10.1145/1321631.1321661>.
- [185] Robbert van Renesse, Yaron Minsky, and Mark Hayden. “A Gossip-style Failure Detection Service”. In: *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Middleware ’98. The Lake District, United Kingdom: Springer-Verlag, 1998, pp. 55–70. ISBN: 1-85233-088-0. URL: <http://dl.acm.org/citation.cfm?id=1659232.1659238>.
- [186] *Request for Supporting LeakSanitizer*. <https://developercommunity.visualstudio.com/t/support-leaksanitizer/826620>. 2019. (Visited on 05/16/2013).
- [187] Reuters. *U.S. cloud-computing failure could spur up to \$19 billion in losses: Lloyd’s*. <https://www.reuters.com/article/us-cyber-cloud-disruption/u-s-cloud-computing-failure-could-spur-up-to-19-billion-in-losses-lloyds-idUSKBN1FC1UC>. 2018.
- [188] *Running ZooKeeper in Production*. <https://docs.confluent.io/current/zookeeper/deployment.html>.

- [189] Colin Scott et al. “Minimizing faulty executions of distributed systems”. In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 2016, pp. 291–309.
- [190] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: <https://research.google.com/archive/papers/dapper-2010-1.pdf>.
- [191] Vladimir Šor and Satish Narayana Srirama. “A statistical approach for identifying memory leaks in cloud applications”. In: *Proceedings of First International Conference on Cloud Computing and Services Science*. CLOSER ’11. Noordwijkerhout, Netherlands, 2011, pp. 623–628.
- [192] Vladimir Šor et al. “Improving statistical approach for memory leak detection using machine learning”. In: *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. ICSM ’13. Washington, DC, USA, 2013, pp. 544–547.
- [193] *Summary of AWS Direct Connect Event in the Tokyo (AP-NORTHEAST-1) Region*. <https://aws.amazon.com/message/17908/>.
- [194] *Summary of the Amazon EC2 DNS Resolution Issues in the Asia Pacific (Seoul) Region (AP-NORTHEAST-2)*. <https://aws.amazon.com/message/74876/>.
- [195] *Summary of the Amazon Kinesis Event in the Northern Virginia (US-EAST-1) Region*. <https://aws.amazon.com/message/11201/>.
- [196] Xudong Sun et al. “Testing Configuration Changes in Context to Prevent Production Failures”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020, pp. 735–751. URL: <https://www.usenix.org/conference/osdi20/presentation/sun>.
- [197] Lalith Suresh et al. “Stable and Consistent Membership at Scale with Rapid”. In: *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX ATC ’18. Boston, MA, USA: USENIX Association, 2018, 387–399. ISBN: 9781931971447.
- [198] *Task Health Checking and Generalized Checks*. <http://mesos.apache.org/documentation/latest/health-checks>.
- [199] *Anonymized for double-blind review*.
- [200] *The Complete History of AWS Outages*. <https://awsmaniac.com/aws-outages/>.

- [201] *Tuning a database cluster with the Performance Service*. https://docs.datastax.com/en/opscenter/6.1/opsc/online_help/services/tuneClusterPerfService.html.
- [202] Raja Vallée-Rai et al. “Soot - a Java Bytecode Optimization Framework”. In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–. URL: <http://dl.acm.org/citation.cfm?id=781995.782008>.
- [203] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. “Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining”. In: *ACM Trans. Comput. Syst.* 21.2 (2003), 164–206. ISSN: 0734-2071. DOI: 10.1145/762483.762485. URL: <https://doi.org/10.1145/762483.762485>.
- [204] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. “A gossip-style failure detection service”. In: *Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, pp. 55–70.
- [205] Robbert Van Renesse and Fred B Schneider. “Chain Replication for Supporting High Throughput and Availability.” In: *OSDI*. Vol. 4. 91–104. 2004.
- [206] Giuliana Santos Veronese et al. “Efficient byzantine fault-tolerance”. In: *IEEE Transactions on Computers* 62.1 (2011), pp. 16–30.
- [207] Paul Voigt and Axel Von dem Bussche. “The eu general data protection regulation (gdpr)”. In: *A Practical Guide, 1st Ed., Cham: Springer International Publishing* 10.3152676 (2017), pp. 10–5555.
- [208] Carl A. Waldspurger. “Memory Resource Management in VMware ESX Server”. In: *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. OSDI '02. Boston, MA: USENIX Association, 2002. URL: <https://www.usenix.org/conference/osdi-02/memory-resource-management-vmware-esx-server>.
- [209] Wenwen Wang. “MLEE: Effective Detection of Memory Leaks on Early-Exit Paths in OS Kernels”. In: *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX ATC '21. 2021, pp. 31–45. ISBN: 978-1-939133-23-6. URL: <https://www.usenix.org/conference/atc21/presentation/wang-wenwen>.
- [210] *What we learned from Google Cloud's June outage*. <https://techhq.com/2019/08/what-we-learned-from-google-clouds-june-outage/>.

- [211] *Windows Performance Recorder*. <https://docs.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-recorder>.
- [212] Yichen Xie and Alex Aiken. “Context- and Path-Sensitive Memory Leak Detection”. In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE ’13. Lisbon, Portugal: Association for Computing Machinery, 2005, 115–125. ISBN: 1595930140. DOI: 10.1145/1081706.1081728. URL: <https://doi.org/10.1145/1081706.1081728>.
- [213] Guoqing Xu and Atanas Rountev. “Precise Memory Leak Detection for Java Software Using Container Profiling”. In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE ’08. Leipzig, Germany: Association for Computing Machinery, 2008, 151–160. ISBN: 9781605580791. DOI: 10.1145/1368088.1368110. URL: <https://doi.org/10.1145/1368088.1368110>.
- [214] Tianyin Xu et al. “Early Detection of Configuration Errors to Reduce Failure Damage”. In: *Proceedings of the The 12th USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’16. Savannah, GA, USA, 2016.
- [215] *Yarn-4254: Accepting unresolvable NM into cluster causes RM to retry forever*. <https://issues.apache.org/jira/browse/YARN-4254>.
- [216] Nofel Yaseen et al. “Aragog: Scalable Runtime Verification of Shardable Networked Systems”. In: *14th USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’20. USENIX Association, 2020, pp. 701–718. ISBN: 978-1-939133-19-9. URL: <https://www.usenix.org/conference/osdi20/presentation/yaseen>.
- [217] Ding Yuan et al. “Be conservative: Enhancing failure diagnosis with proactive logging”. In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 2012, pp. 293–306.
- [218] Ding Yuan et al. “Sherlog: error diagnosis by connecting clues from run-time logs”. In: *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 2010, pp. 143–154.
- [219] Ding Yuan et al. “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. Broomfield, CO: USENIX Association, 2014,

- pp. 249–265. ISBN: 978-1-931971-16-4. URL: <http://dl.acm.org/citation.cfm?id=2685048.2685068>.
- [220] Xinhao Yuan and Junfeng Yang. “Effective Concurrency Testing for Distributed Systems”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, 1141–1156. ISBN: 9781450371025. DOI: 10 . 1145 / 3373376.3378484. URL: <https://doi.org/10.1145/3373376.3378484>.
 - [221] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: <http://dl.acm.org/citation.cfm?id=1863103.1863113>.
 - [222] Ennan Zhai et al. “Check before You Change: Preventing Correlated Failures in Service Updates.” In: *NSDI*. 2020, pp. 575–589.
 - [223] Yongle Zhang et al. “Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: ACM, 2017, pp. 19–33. ISBN: 978-1-4503-5085-3. DOI: 10 . 1145 / 3132747.3132768. URL: <http://doi.acm.org/10.1145/3132747.3132768>.
 - [224] Yongle Zhang et al. “The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, 131–146. ISBN: 9781450368735. DOI: 10 . 1145 / 3341301 . 3359650. URL: <https://doi.org/10.1145/3341301.3359650>.
 - [225] Yongle Zhang et al. “Understanding and Detecting Software Upgrade Failures in Distributed Systems”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP ’21. Virtual Event, Germany: Association for Computing Machinery, 2021, 116–131. ISBN: 9781450387095. DOI: 10 . 1145 / 3477132 . 3483577. URL: <https://doi.org/10.1145/3477132.3483577>.
 - [226] Yongle Zhang et al. “Understanding and Detecting Software Upgrade Failures in Distributed Systems”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, 2021, 116–131. ISBN: 9781450387095. URL: <https://doi.org/10.1145/3477132.3483577>.

- [227] *ZooKeeper-1208: Ephemeral node not removed after the client session is long gone.* <https://issues.apache.org/jira/browse/ZOOKEEPER-1208>.
- [228] *ZooKeeper-2201: Network issue causes cluster to hang due to blocking I/O in synch.* <https://issues.apache.org/jira/browse/ZOOKEEPER-2201>.
- [229] *ZooKeeper-2319: UnresolvedAddressException cause the listener exit.* <https://issues.apache.org/jira/browse/ZOOKEEPER-2319>.
- [230] *ZooKeeper-2325: Data inconsistency when all snapshots empty or missing.* <https://issues.apache.org/jira/browse/ZOOKEEPER-2325>.
- [231] *ZooKeeper-2774: Ephemeral znode will not be removed when session timeout, if the system time of ZooKeeper node changes unexpectedly.* <https://issues.apache.org/jira/browse/ZOOKEEPER-2774>.
- [232] *ZooKeeper-3131: WatchManager resource leak.* <https://issues.apache.org/jira/browse/ZOOKEEPER-3131>.
- [233] *ZooKeeper-3144: Potential ephemeral nodes inconsistent due to global session inconsistent with fuzzy snapshot.* <https://issues.apache.org/jira/browse/ZOOKEEPER-3144>.
- [234] *ZooKeeper-3531: Synchronization on ACLCache cause cluster to hang when network/disk issues happen during datatree serialization.* <https://issues.apache.org/jira/browse/ZOOKEEPER-3531>.
- [235] *ZooKeeper-914: QuorumCnxManager blocks forever.* <https://issues.apache.org/jira/browse/ZOOKEEPER-914>.