UNIVERSITY OF CALIFORNIA, SAN DIEGO

Hardening Cloud and Datacenter Systems against Misconfigurations: Principles and Tool Support

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Tianyin Xu

Committee in charge:

Professor Yuanyuan Zhou, Chair
Professor Pamela C. Cosman
Professor William G. Griswold
Professor Scott R. Klemmer
Professor Stefan R. Savage
Professor Geoffrey M. Voelker

2017

DEDICATION

To those who gave me a chance when no one else would.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

The journey of a Ph.D. is a voyage across the Narrow Sea. One cannot arrive the Seven Kingdom without the guidance and support from advisors, mentors, colleagues, friends, and family. At this moment of landing and resailing, I am incredibly grateful to all the individuals who helped me along this journey.

First and the foremost, I want to thank my advisor, Yuanyuan (YY) Zhou, for her unwavering guidance, support, and inspiration throughout my Ph.D. study, without which I could never have dreamed my dream. I don't know how to express my gratefulness to her—she completely changed my life! YY is the best advisor I can ever dream of. She guided me to walk down the roads to be called a Ph.D.; she enlightened all the technical insights that form the fundation of this dissertation; she exposed multiple career choices and created the opportunities for me to experience. In fact, what I learned from her is far beyond a degree, a dissertation, and a job, but the way to think, to learn, and to do research; most importantly, to be someone I want to be. I can only hope to emulate (or at least imitate) her creativity, vision, and professionalism, and mostly her caring about students' happiness and success, as I myself start being an advisor.

I would like to thank Geoff Voelker for doing so many special things for me. Geoff always helps me and saves me when I am in trouble—he helped all my important talks and gave me countless advice on presentation, teaching, and research. I cannot forget that he revised every slide of my job talk and encouraged me to practice, when I was extremely frustrated and intimidated. Geoff exemplifies what means to be a true professor who sets the goal of helping students; who explains complex materials in an incredibly graceful way; who always makes classes and labs fun. My goal as a professor is to be an educator like him to focus on and make positive impact to the next generation.

I am also extremely grateful to (and honored by) the other members of the committee. Stefan Savage has always been the lighthouse I look up to. I have learned so much from him and been so much inspired by the way he picks (often crazy) research problems, articulates the fundamental questions, and makes impact through perspectives and solutions that can make a difference. Stefan wrote my letter (for job search) during his honeymoon, which I can never forget. Scott Klemmer shaped my understanding on Human Computer Interaction (HCI) and taught me how to look at misconfigurations from an HCI perspective. I am significantly influenced by his style of thinking and articulating. I learned Software Engineering from Bill Griswold, which has influenced my philosophy of engineering software systems. I should have failed to propose my disseration (let alone complete it), if Pam Cosman was not willing to try me out.

I did two fruitful summer internship at NetApp working with Shankar Pasupathy. Shankar is the best mentor and collaborator I can imagine, who always gives me the freedom of exploration, as well as data and systems support. We have extensively collaborated in the past six years, which constructs the major parts of this dissertation.

I was extremely lucky to work with many exceptional people in the Operating Systems Group (Opera) during my time at UCSD. It is difficult to imagine undertaking this dissertation effort without the experiences I shared with them. I want to thank Ding Yuan, Soyeon Park, Xiao Ma, Weiwei Xiong, Tianwei Sheng, Jiaqi Zhang, and Ryan Huang for setting up great examples and for the mentoring I received from them. I want to thank Ding, Ryan, Shan Lu, and Lin Tan for sharing their experiences of being a faculty, and most importantly, paved the way for me as such great examples in academia. Shan called me when I was about to quit (to join a startup). Her perspective of learning and self-improvement is inspirational. Ding and Ryan gave me countless advice on almost everything about research and career. Never a single time they turn me down when I have questions or doubts. Weiwei and Tianwei have always been my big brothers who look after me and my family. I also want to thank Xinxin Jin for being such a wonderful fellow. The peer pressure we gave each other turns out to be incredibly beneficial. We together experienced the up and down, and eventually survived and strived. It has been a privilege to collaborate with Ding, Jiaqi, Ryan, Jing Zheng, Tianwei, Xuepeng Fan, Long Jin, Xinxin, Tao Cai, Liqiong Yang, Le Lu, and Han Min Naing from Opera. None of my projects could be accomplished without their incessant and selfishless contributions. I also thank other members in Opera, including Zuoning Yin, Michael Lee, Robert Liu, Rishan Chen, Gen Liu, Zhuoer Wang, Chengcheng Xiang, and Yudong Wu. They and all the other members make Opera a family for me and my wife, even outside UCSD. Rishan looked after me when I was evicted by a landlord. I always look up to his courage to honestly and bluntly pursue dreams.

I am also thankful to many other people at UCSD for teaching me, enlightening me, and helping me, especially Sanjoy Dasgupta, Philip Guo, Sorin Lerner, Alex Snoeren, Aaron Schulman, and George Varghese. Sanjoy took the risk on me when I applied for Teaching Assistant (TA) for the first time (clearly I was an awkward TA at that time). I learned so much from Sanjoy who demonstrated the magic of explaining complex algorithms neatly. Alex always asks sharp questions to push me to think more and deeper. Sorin, Philip, and Aaron helped me shape my job talk, which composes the flow of this dissertation. Specially, I want to thank Geoff, Stefan, Alex, and other sysnet faculties for making sysnet a caring family. I want to thank Julie Conner and Jennifer Folkestad for always being so helpful and patient on tedious administrative stuffs.

I want to thank all my other friends at UCSD for making my Ph.D. journey so enjoyable, especially Nathan Farrington, Yupeng Fu, Danny Huang, Yanqin Jin, Rishi Kapoor, Vector Li, Zack Lipton, Lonnie Liu, Feng Lu, Kian Win Ong, Vineet Pandey, Quentin Pleplé, Matus Telgasky, Malveeka Tewari, Shelby Thomas, David Wang, Jianguo Wang, Zhaomo Yang, and Yiying Zhang. In particular, Matus has always been a mentor and given me many meta-level advice on research and teaching. I am so happy that we will continue to be colleagues at UIUC. Kian Win helped me estabalish the connection with Facebook where I will spent my “prebbatical.” I enjoyed TA for eight quarters and had the privileges to work with and learn from many excellent fellow TAs, especially Matus, Quentin, Xinxin, Kacy Espinoza, Karthikeyan Balasubramaniam, Kabir Gogia, Daniel Knapp, Hung Nguyen, Saurabh Chandra, and Zhaomo.

I want to thank my other colleagues at NetApp and Whova where I did summer internships, especially Lakshmi Bairavasundaram, Congming Chen, Yuelu Duan, Daniel Fryer, Art Harkin, Raghavan Kalkunte, Deepak Kenchammana-Hosekote, Xing Lin, Laurent Nicolas, Kevin Nomura, Anusha Perumal, Gokul Soundararajan, Sethuraman Subbiah (my buddy), Rukma Talwadker, Suli Yang, and Junji Zhi.

I want to thank all my other collaborators outside UCSD whom I had the priviledge to work with. Specially, I want to thank Zhenhua Li for trusting me and teaching me so many things. Our collaboration is fruitable, and more than that is the joy and bitterness we have gone through together, with Schopenhauer and Nietzsche.

I want to thank Martha Stacklin at Center for Teaching Development. I benefit tremendously from her classes where I developed my taste and style of teaching and public speaking. Martha later encouraged me to participate the Grad Slam completition and taught me how to present in front of non-CS audience. I thank Carolyn Passeneau for being my English tutor who helped me improve my English as an international student with terrible TOEFL scores. She is well deserved the EIA Tutor of the Year in 2012.

Last but not the least, I am deeply grateful to my family. I want to thank my parents for their unconditional love and support. My mom has always been my inspiration from whom I learned passion and work ethics. I do not talk to them often but I know they love me. I thank my wife Fei for her boundless and unfading love. She is everything to me. Not a single day goes by that I am not reminded of how lucky I am to have her as a partner and a best friend. We share every piece of life together and always love it for what it is. I also thank my parents-in-law for their understanding and support.

My Ph.D. study is supported by grants from NSF and NetApp.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the 24th ACM Symposium on Operating Systems Principles, 2013. Xu, Tianyin; Zhang, Jiaqi; Huang, Peng; Zheng, Jing; Sheng, Tianwei; Yuan, Ding; Zhou, Yuanyuan; Pasupathy, Shankar. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, 2016. Xu, Tianyin; Jin, Xinxin; Huang, Peng; Zhou, Yuanyuan; Lu, Shan; Jin, Long; Pasupathy, Shankar. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2015. Xu, Tianyin; Jin, Long; Fan, Xuepeng; Zhou, Yuanyuan; Pasupathy, Shankar; Talwadker, Rukma. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in ACM Computing Surveys, Vol. 47, No. 4, Article 70, 2015. Xu, Tianyin; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

VITA

2000–2007	Bachelor of Science, Computer Science Nanjing University, China
2007–2010	Master of Engineering, Computer Engineering Nanjing University, China
2011–2017	Doctor of Philosophy, Computer Science University of California, San Diego

PUBLICATIONS

Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. How Do System Administrators Resolve Access-Denied Issues in the Real World? In *Proceedings of the 35th Annual CHI Conference on Human Factors in Computing Systems (CHI'17)*, Denver, CO, May 6-11, 2017.

Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, Nov. 2-4, 2016.

Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, Bergamo, Italy, Aug. 31-Sep. 4, 2015.

Tianyin Xu and Yuanyuan Zhou. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys*, Volume 47, Number 4, Article 70, Jul. 2015.

Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, Farmington, PA, Nov. 3-6, 2013.

Peng Huang, **Tianyin Xu**, Xinxin Jin, and Yuanyuan Zhou. DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior. In *Proceedings of the 14th International Conference on Mobile Systems, Applications, and Services (MobiSys'16)*, Singapore, Singapore, Jun. 26-30, 2016.

Xinxin Jin, Peng Huang, **Tianyin Xu**, and Yuanyuan Zhou. NChecker: Saving Mobile App Developers from Network Disruptions. In *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys'16)*, London, UK, Apr. 18-21, 2016.

Zhenhua Li, Weiwei Wang, **Tianyin Xu**, Xin Zhong, Xiang-Yang Li, Yunhao Liu, Christo Wilson, and Ben Y. Zhao. Exploring Cross-Application Cellular Traffic Optimization with Baidu TrafficGuard. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, Santa Clara, CA, Mar. 16-18, 2016.

Zhenhua Li, Christo Wilson, **Tianyin Xu**, Yao Liu, Zhen Lu, and Yinlong Wang. Offline Downloading in China: A Comparative Study. In *Proceedings of the 15th ACM Internet Measurement Conference (IMC'15)*, Tokyo, Japan, Oct. 28-30, 2015.

Qinhui Wang, Baoliu Ye, Bin Tang, **Tianyin Xu**, Song Guo, Sanglu Lu, and Weihua Zhuang. ALETHEIA: Robust Large-Scale Spectrum Auctions against False-Name Bids. In *Proceedings of the 16th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'15)*, Hangzhou, China, Jun. 22-25, 2015.

Zhenhua Li, Cheng Jin, **Tianyin Xu**, Christo Wilson, Yao Liu, Linsong Cheng, Yunhao Liu, Yafei Dai, and Zhi-Li Zhang. Towards Network-level Efficiency for Cloud Storage Services. In *Proceedings of the 14th ACM Internet Measurement Conference (IMC'14)*, Vancouver, Canada, Nov. 5-7, 2014.

Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, **Tianyin Xu**, and Yuanyuan Zhou. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, Salt Lake City, UT, Mar. 1-5, 2014.

ABSTRACT OF THE DISSERTATION

Hardening Cloud and Datacenter Systems against Misconfigurations: Principles and Tool Support

by

Tianyin Xu

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Yuanyuan Zhou, Chair

Misconfigurations (a.k.a., configuration errors from a system's standpoint) are among the dominant causes of today's catastrophic system failures that turn down cloud-scale services and affect hundreds of millions of end users. Despite their wide adoption, traditional fault-tolerance and failure-recovery techniques are not effective in dealing with configuration errors, especially in large-scale software systems deployed in cloud and datacenters. To make the matters worse, even the tolerance and recovery mechanisms themselves are often misconfigured in the real world, which impairs the immune system of the entire cloud and datacenters.

This dissertation explores two fundamental questions towards the solutions for the inevitable misconfigurations—how to build reliable cloud and datacenter systems in the face of configuration errors; moreover, how to prevent misconfigurations in the first place by better configuration design. The goal is to enable software systems to *proactively* anticipate and defend against misconfigurations, rather than reacting to their manifestations and consequences.

This dissertation presents three key principles of systems design and implementation for hardening cloud and datacenter systems against misconfigurations—anticipating misconfigurations, early detection of configuration errors, and simplicity-oriented configuration design. The dissertation demonstrates that applying these principles can effectively defend cloud and datacenter systems against misconfigurations. Moreover, the dissertation presents the corresponding techniques and tool support that can automatically and systematically apply these principles to existing systems software.

The main technical insight is that configurations are essentially used by the systems, while configuration errors are mostly manifested through the faulty execution that uses erroneous configuration values. Therefore, by analyzing the system's code that uses configuration values, one can understand and make use of system-level information of configurations to build defense against potential errors. This dissertation first presents SPEX that enables systems to anticipate misconfigurations. SPEX automatically infers configuration constraints from a system's source code, and then leverages the constraints to test the system's resilience to misconfigurations and detect error-prone configuration design/handling. On step further, the dissertation introduces PCHECK to automatically generate checking code which captures configuration errors at the system's initialization phase to prevent their late manifestations and the corresponding failure damage.

Going beyond, this dissertation presents simplicity-oriented configuration design towards more usable and less error-prone software configuration. The key idea is to apply the user-centric philosophy to design configuration as an interface—configurations are essentially the interface for controlling and customizing system behavior, but have rarely been treated as it is. The dissertation shows that configurations in today's systems software can be significantly simplified and effectively navigated, with the understanding of how they are actually used in the field.

Chapter 1

Introduction

"To err is human; to forgive, divine."
—Alexander Pope

Misconfigurations (a.k.a., configuration errors from a system's standpoint) are among the dominant causes of today's catastrophic system failures that turn down cloud-scale services and affect hundreds of millions of end users. Barroso et al. show that misconfigurations are the second largest cause of service disruptions at one of Google's main services [20]. Maurer reveals that misconfigurations are among the most common "pathologies" that amplify failures and cause them to widespread in Facebook [97]. Specifically, misconfigurations haven been reported as one major cause of failures in a variety of system components deployed in cloud and datacenters, including storage systems [12,77,185], data-intensive computing frameworks [56,125,187], database systems [56, 112, 185], and network infrastructure [87, 96, 151]. In recent years, almost every cloud company (e.g., Google, Facebook, Microsoft, Amazon, LinkedIn) has experienced outages and downtime induced by misconfigurations, affecting millions of their customers and causing massive revenue loss [32,57,94,152,155,158].

Note that almost all the aforementioned systems are built with the mindset of fault tolerance and failure recovery, as component failures (e.g., due to hardware faults and software defects) are a norm in today's large-scale, rapid-changing software systems deployed in cloud and datacenters [20, 97]. Fault tolerance is mostly achieved by redundancy. For example, RAID and erasure code are commonly used to cope with machine/disk failures [52, 70, 82]; backup and replication are widely deployed to deal with software component failures upon which fail-over would be triggered [11, 34, 36, 55]; redundant paths and fallback protocols are designed to handle path/route failures [29, 40, 54]. In terms of recovery, rebooting and rollback based techniques are commonly adopted [35, 116]. Unfortunately, these mechanisms are less effective in handling misconfigurations [97, 113]. In cloud and datacenter systems, a single configuration error is often replicated to hundreds or even thousands of nodes, which significantly enlarges the impact of the error and makes redundancy-based fault tolerance ineffective. Moreover, as the errors typically reside in persistent configuration files or databases, they are resistant to rebooting-based recovery techniques. To make matters worse, even fault tolerance and recovery themselves are often misconfigured in the real world, which impairs the immune system of the entire cloud and datacenter and thus lead to catastrophes, as reported by many newsworthy outages [32, 142, 152, 157, 158].

Given the severity and prevalence of misconfigurations, there has been a wealth of literature that aims at tackling misconfigurations introduced in the field, mostly with the focus on detecting misconfigurations in the field [22, 41, 50, 71, 92, 115, 118, 131, 189, 190] and troubleshooting their consequences in terms of failures and anomalies [6, 16–18, 101, 123, 149, 150, 165, 168, 170, 186, 191]. While these efforts provide useful, effective remedies for misconfigurations and/or their consequences, they share the following fundamental limitations:

- they are *reactive* to misconfigurations and thus can hardly prevent failures in the first place. In fact, despite the progress in misconfiguration troubleshooting, diagnosing failures in cloud and datacenter systems still remains one of the most challenging tasks and the effectiveness largely relies on the skills and experiences of human administrators, operators, and engineers.[1] Often, when a failure occurs in the field, it is even hard to determine if the root cause is a configuration error or a bug or something else [172, 174]. As shown in Chapter 2, configuration errors could lead to similar symptoms as bugs such as crashes, hangs, and dysfunctions. It is not uncommon that diagnosis can take hours or even days [29, 54, 169].

- they fix symptoms instead of roots. As revealed in Chapter 2, the same configuration error could be introduced again and again by different operators in different systems and scenarios (Figure 2.1 is such an example). As pointed out by Lampson, unlike code which is written once, configurations could be different for different use cases and could be done by less skilled people (c.f., Appendix A) based on documentation that is usually voluminous, obscure, and incomplete [88].

- they are not designed for cloud and datacenter systems. Many of the proposed detection and troubleshooting approaches are based on applying machine learning with the requirements of large configuration datasets [6, 22, 41, 101, 115, 131, 165, 168, 186, 189, 190], which does not fit the cloud and datacenter model (refer to §3.2.2 for the detailed discussion); some other approaches require instrumenting source code in order to record runtime execution traces [16, 18, 191, 192] which could be challenging when being applied to distributed cloud and datacenter systems at scale.

Today, misconfigurations are still being treated among the most thorny systems problems. For example, Welsh advocates "an escape from configuration hell" as the first problem among what he wishes systems researchers would work on [169]. Gunawi et al. call for research in the cloud community to deal with configuration issues [56].

[1]In different operation disciplines (e.g., traditional system administration, DevOps, and site reliability engineering), different types of personnel are responsible for managing configurations. Appendix A discusses these operation disciplines and their implications to this dissertation. In the rest of this dissertation, we use "operators" to refer to personnel who operate cloud and datacenter systems and thus are responsible for system configurations despite their actual job titles.

This dissertation proposes the perspective of hardening software systems' own defense against misconfigurations, as a more fundamental solution. Instead of relying on external tools and procedures, this dissertation investigates *how to build reliable cloud and datacenter systems in the face of configuration errors* and *how to prevent misconfigurations in the first place by better configuration design*. The goal of this dissertation research is to enable software systems to *proactively* anticipate and defend against misconfigurations, rather than reacting to their manifestations and/or consequences.

1.1 A Systems Perspective

From a system's view, the manifestation of configuration errors are not too much different from other types of systems errors, such as software bugs that have been extensively studied in the past decades. At a high level, configuration errors are erroneous values residing in the configuration files or databases (e.g., Windows Registry). These values are read into the software systems at the system's startup time and stored in the corresponding program variables or data structures [Chapter 2 (§2.3.2) provides detailed descriptions and examples of this process]. When the system needs to apply a configuration value, it directly uses the program variable that stores the value. For example, if the configuration value specifies the path of a file `f` on the local file system, which is stored in a program variable `p`, the system would `open(p)` when it needs to read or write `f`. However, if the value is erroneous, the usage would fail. In this example, if `f` does not exist on the file system due to misspelling or the program cannot access `f` due to lack of privileges, the `open` call would return `-1` with `errno` encoding the root cause. Chapters 2 and 3 give a number of examples that demonstrate how configurations are used in source code and how the errors are manifested during the runtime execution.

Note that configuration errors cannot be detected or diagnosed using traditional bug detection or diagnosis techniques. This is because they are not defects in the code introduced by developers; instead, they are erroneous settings in the configuration files introduced by operators in the field. Given the enormous configuration space (Chapter 4 quantifies the complexity in details), it is challenging for traditional bug detection or in-house testing approaches to enumerate all potential misconfigurations. Nevertheless, as discussed in Chapters 2 and 3, certain "correct" system behavior in the traditional sense, such as crashing and fail-stop, is not acceptable when dealing with misconfigurations.

The impact of configuration errors depends on how a system treats its configurations. If the system proactively checks its configuration values at the startup time and reports errors in log messages that pinpoint the bad configuration values, operators can fix the errors in time and prevent failures. On the other hand, if the system assumes correct configurations and directly uses the value on demand (when it absolutely needs it), the configuration errors would lead to disasters, as revealed in Chapter 3. Furthermore, even the system does capture a configuration error; if it fails to provide pinpointing messages but crashes or terminates silently (which is not uncommon as shown in Chapter 2), it is extremely difficult for operators to understand the root causes and to fix the misconfigurations efficiently. In terms of the manifestations, there is no fundamental difference between configuration errors and software bugs because both of them could be manifested through crashes, hangs, exceptions, error code, etc.

On the other hand, software developers' attitude towards misconfigurations is often quite different from how they treat bugs. For software bugs, the developers typically take a responsible and active role. This is reflected in many ways, such as various choices of bug-tracking databases, patch releases, unit/regression tests, and bug checkers. In contrast, developers often take laid-back roles in handling misconfigurations, because "they are operators' faults." Such an attitude is reflected in two main aspects: (1) misconfigurations are much less rigorously tracked; (2) after a misconfiguration is identified as the root cause, developers often do not take any further actions, such as changing the code or releasing patches to avoid the same misconfiguration being introduced by other operators (which is often the case). In fact, taking an active role in handling misconfigurations benefits developers themselves. For example, enabling systems to detect and pinpoint misconfigurations can reduce the number of issues reported to the developers, saving their precious time. Essentially, different from software bugs, misconfigurations can be directly fixed by operators in the field.

Thinking about (mis)configurations from the systems perspective also brings tremendous opportunities for the solutions. As any configurations are essentially used by the systems; hence, by analyzing the system's code that uses configuration values, one can automatically and systematically obtain a lot of system-level knowledge of configurations, such as systems constraints of configuration values and manifestation patterns of configuration errors. Such knowledge is extremely valuable—it enables evaluating configuration handling, exposing misconfiguration vulnerabilities inside the systems, and generating configuration checking code for the systems, as demonstrated in Chapters 2 and 3. Furthermore, thinking and designing configurations as a system's interface (for controlling and customizing systems behavior) can significantly improve the usability and reduce the error-proneness of configurations, as shown in Chapter 4. After all, many misconfigurations are derived from human errors and mistakes.

1.2 Dissertation Contributions

This dissertation presents three key principles of systems design and implementation for hardening cloud and datacenter systems against misconfigurations. These principles include *anticipating misconfigurations* (Chapter 2), *early detection of configuration errors* (Chapter 3), and *simplicity-oriented configuration design* (Chapter 4). The dissertation demonstrates that these principles can effectively help cloud and datacenter systems defend against real-world misconfigurations. Furthermore, the dissertation presents the corresponding techniques and tool support that automatically and systematically applies these principles to enable the defense for existing systems software. The following outlines how this dissertation addresses the contributions.

1.2.1 Anticipating Misconfigurations with SPEX

Chapter 2 reveals that many of today's mature software systems widely deployed in cloud and datacenters do not anticipate and thus are vulnerable to misconfigurations. As a result, misconfigurations could often lead to disastrous system behavior such as crashes, hangs, and silent termination. Failure diagnosis is often misled by such perplexing behavior, causing unnecessarily long repair time.

To enable cloud and datacenter systems to anticipate and defend against misconfigurations, Chapter 2 presents SPEX to automatically infer configuration requirements (termed *constraints*) from source code and then use the inferred constraints to (1) expose misconfiguration vulnerabilities (bad system reactions such as crashes, hangs, and silent failures); and (2) detect certain types of error-prone configuration handling.

SPEX's insight is that many configuration constraints are reflected in the system's source code, and can be automatically inferred via static code analysis, based on the properties of operations and system/library APIs that use configuration values. SPEX tracks the data-flow of configuration values in the source code, and looks for patterns of various kinds of configuration constraints, including data types, data ranges, control dependencies, and value relationships. With the inferred constraints, a testing tool is built based on configuration-error injection to expose misconfiguration vulnerabilities—it intentionally violates the inferred constraints to generate misconfigurations, and tests how the system reacts (bad reactions are recorded and reported to developers). Moreover, the inferred constraints can be analyzed to detect error-prone configuration handling, namely including inconsistency, silent overruling, and unsafe behavior.

SPEX has uncovered 743 misconfiguration vulnerabilities of various types and 112 error-prone configuration handling cases in both commercial and open-source systems. So far, about half have been fixed or confirmed by the corresponding developers.

1.2.2 Enforcing Early Detection with PCHECK

Chapter 3 shows that misconfigurations of fault tolerance and error handling are particularly dangerous. Since these configurations are not needed at initialization or during normal operations, many systems do not check their values early but directly use the values under critical circumstance (when encountering faults/errors). Thus, the errors become latent until their manifestations cause catastrophes. For such latent errors, early detection is the key to reducing failure damage. However, Chapter 3 reveals that even in widely-deployed cloud and datacenter systems, many critically important configurations (e.g., those for fault tolerance) do not have any initial checking code to validate the correctness of the settings, and thus are subject to latent errors.

To enable early detection of configuration errors, Chapter 3 presents PCHECK to automatically generate configuration checking code and invoke the checking at the system's initialization phase. PCHECK exploits the fact that the actual code that uses configuration values (which already exists in source code) can serve as an implicit form of checking (e.g., opening a file path specified by a configuration value implies a capability check). Such usage-implied checking precisely captures how the configuration values should be used in actual program execution. PCHECK generates checking code by emulating the late execution that will use the configuration values; meanwhile capturing any anomalies exposed during the emulated execution as the evidence of configuration errors—same anomalies would occur in real execution, if the errors are not fixed.

The key challenges of PCHECK is to make the checking code effective and safe. PCHECK statically extracts instructions that transform, propagate, and use configuration values from the system program. To execute the extracted instructions, PCHECK makes the best effort to determine the values of dependent variables and produce self-contained execution context. To ensure that the checking code is safe to invoke, PCHECK sandboxes the emulated execution by instruction rewriting to prevent any side effects on the running system or its environment. Furthermore, PCHECK inserts instructions to capture the anomalies that may occur during the emulated execution, based on which it reports errors. PCHECK can detect over 75% of real-world latent configuration errors at system initialization time, and is generally applicable to cloud and datacenter systems because it does not rely on any predefined rules or external datasets.

1.2.3 Simplicity-oriented Configuration Design

One fundamental reason for today's prevalent misconfigurations is the tremendous but still-increasing complexity of configuration, reflected by hundreds or even thousands of configuration parameters ("knobs") exposed by cloud and datacenter systems. Many knobs have various constraints, consistency requirements, and dependencies with other knobs. Such complexity makes it daunting and error-prone for operators to configure cloud and datacenter systems.

Chapter 4 presents design principles and disciplines to make configuration more usable and less error-prone, with a user-centric design philosophy. It starts by questioning whether the complexity is indeed necessary based on the study of real-world configuration usage characteristics: Do operators really need so many knobs? Can they manage the complexity? What are their common difficulties and mistakes? The answers to these questions conclude that configurations are commonly over-designed—only a small percentage of configuration knobs are set by the majority of operators in the field, while the majority of knobs are seldom touched. Many knobs are neither necessary nor worthwhile—they make configuration more complex but produce little benefit. On the other hand, complexity does come with a cost: operators often have difficulties in finding the right knob(s) to achieve intended system behavior or performance goals; worse, the excessive complexity prevents operators from understanding the configurations thoroughly and examining the settings carefully.

Based on these understandings, Chapter 4 presents a few concrete, practical design guidelines which could significantly reduce the configuration space and thus simplify configuration design, with little impact on the flexibility desired by operators. To help navigate the vast configuration space in existing software, the chapter also introduces COX to help operators find the right knobs by expressing their intent based on natural language processing, and shows that leveraging the field characteristics of configuration usage can greatly improve the effectiveness of configuration navigation.

1.3 Dissertation Scope

This dissertation focuses on systems software deployed in cloud and datacenters, such as server software, storage software, and computing infrastructure. It does not touch configurations of network devices [23, 24, 51, 54, 151], desktop software [26, 78, 86], or mobile apps [73, 76, 80, 93, 160]. Misconfigurations of such are also important problems; however, they have fundamentally different characteristics from those in cloud and datacenter software and thus require different solutions. Moreover, this dissertation primarily focuses on misconfigurations that violate the correctness of the software with less concerns on performance and security (unless they cause severe usability issues). Lastly, similar as prior work, it focuses on misconfigurations of parameters rather than those of software components or hardware. It is reported that parameter misconfigurations account for the majority of real-world configuration issues [185].

Chapter 2

Anticipating Misconfigurations

"*Perfection is achieved on the point of collapse.*"
—C. N. Parkinson

The first and foremost principle towards building reliable software systems in the face of configuration errors is to anticipate misconfigurations and accept that they are inevitable. Unfortunately, while we software developers are often trained and educated to implement our systems to tolerate hardware faults and network errors, there is less emphasis on tolerating or reacting gracefully to misconfigurations in the field. In fact, just like hardware faults, misconfigurations (many of which stem from human errors and mistakes) are a force of nature, too. In reality, developers often unconsciously assume correct configurations. As a result, many configuration errors lead to system crashes, hangs, incorrect results, etc. Such disastrous and perplexing behavior leaves operators clueless, unnecessarily increasing the difficulties of troubleshooting and failure diagnosis. On the other hand, if the system could explicitly pinpoint the configuration errors with clear log messages, guided by such messages, operators can directly fix their misconfigurations by themselves, without resorting to developers. Essentially, *different from software bugs, misconfigurations can be directly fixed by operators themselves if precise error messages are provided by the system.*

2.1 Introduction

This chapter presents the principles and tool support that enable software systems to anticipate and defend against misconfigurations. Specifically, it aims at improving configuration design and implementation of today's software systems by (1) enabling the systems to react gracefully to configuration errors (e.g., pinpointing the erroneous configuration parameters and their values); (2) improving configuration design and handling to make them more usable and less error-prone.

Achieving the above goals would need the specifications of configuration requirements, referred to as *configuration constraints*. A constraint of a configuration parameter specifies the data type, format, value range, dependency and correlation with other parameters, etc., which forms the correctness definition of its value. Since large-scale systems usually contain hundreds or even thousands of configuration parameters, it is time-consuming and error-prone to let developers specify each constraint manually [84]. One potential solution is to leverage documentation (e.g., user manuals). Unfortunately, documentation is written in natural languages, and thus is hard to analyze automatically. Moreover, documentation is often incomplete and outdated, as reported in prior studies [88, 124].

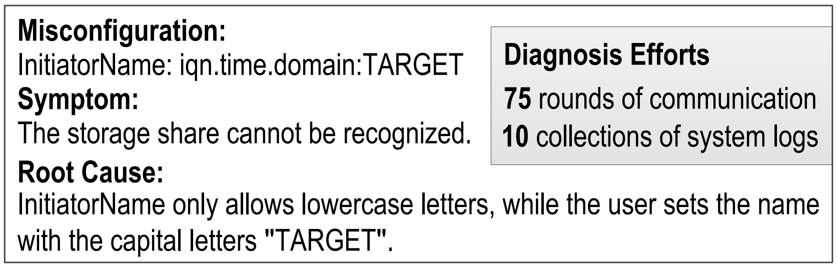
As source code always contains up-to-date information, the idea is to automatically infer configuration constraints, including types, ranges and cross-parameter dependencies from source code, by analyzing how the values of configuration parameters are actually read and used. This idea is grounded in a tool called SPEX. Furthermore, SPEX leverages the automatically inferred constraints to build two use cases: (1) exposing bad system reactions to configuration errors (e.g., crashes, hangs, and dysfunctions) through injecting misconfigurations that violate the constraints; (2) detecting certain types of error-prone configuration design and handling.

2.2 Background

One of the reasons that today's software systems do not anticipate misconfigurations well is our (software developers') attitude towards misconfigurations, which is quite different from how we treat software bugs. For bugs, developers typically take a responsible and active role. This is reflected in many ways, such as various choices of bug-tracking databases, patch releases, unit/regression tests, and bug checkers. In contrast, developers often take much laid-back roles in handling misconfigurations, because "they are operators' faults." Such an attitude is reflected in at least two main aspects: (1) misconfigurations are less rigorously tracked; (2) after a misconfiguration is discerned as the root cause, developers often do not take further actions, e.g., changing code or releasing patches to avoid the same misconfigurations being introduced again (which is often the case in reality).

In most cases, even though it is the operators who introduced the erroneous values, they should not take all the blame. After all, a misconfiguration is referred to as an "error" simply because it does not match our (software developers') requirements for configuration. Therefore, before blaming operators for configuration errors, we need to question whether we have the right requirements in the first place. Are we assuming too much from operators? Operators do not write our code and sometimes cannot read our code. How could they have the same level of accurate understanding of the requirements and impact of various configuration values as we do? Are our configuration requirements too strict or too confusing? After all, operators are also human beings, and just like us, also make mistakes, especially when the requirements are error-prone.

In fact, misconfigurations affect not only operators, but also software developers, because they need to spend time and effort in troubleshooting and correcting them. A recent study [185] shows that configuration issues account for 27% of customer support



Misconfiguration:
InitiatorName: iqn.time.domain:TARGET
Symptom:
The storage share cannot be recognized.
Root Cause:
InitiatorName only allows lowercase letters, while the user sets the name with the capital letters "TARGET".

Diagnosis Efforts
75 rounds of communication
10 collections of system logs

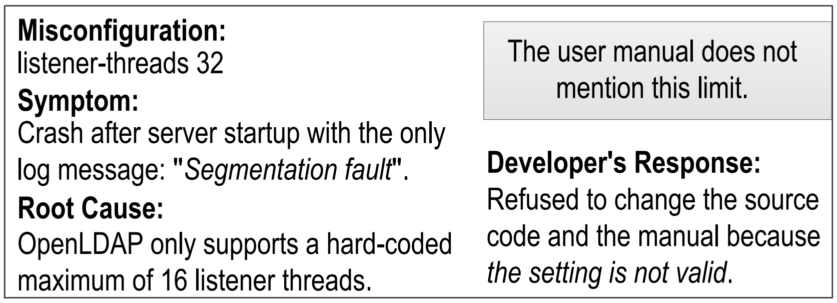
Figure 2.1. A real-world example from a commercial company. The configuration constraint was too strict and multiple operators made the same mistake despite two documents explaining it.

cases in a major storage company. Regardless of the root causes (software bugs or misconfigurations), the system often misbehaves with similar symptoms (e.g., crashes, missing functionalities, and incorrect results). This leaves operators no choice but to report the problems to the technical support. When support engineers are misled by such ambiguous symptoms, the diagnosis often takes unnecessary long time [185].

Figures 2.1 and 2.2 give two real-world examples to further illustrate the above points. As shown in Figure 2.1, a commercial system[1] required operators to type all lowercase letters for the configurations of the initiator names ("`InitiatorName`") of iSCSI adapters. This requirement is too strict. As a consequence, several operators made the same mistakes and had to call the company to help troubleshoot the problem. In this particular case, the diagnosis took over 75 rounds of communication with the customer, as well as 10 rounds of debugging message collection. It resulted in not only the customer's downtime but also high supporting cost.

The second example, as shown in Figure 2.2, is from the latest version of OpenLDAP. With the parameter "`listener-threads`" being configured to be larger than 16, the LDAP server would crash after startup, with "segmentation fault." The crash symptom misled at least two operators to report it as a software bug. This problem is detected by our tool. Unfortunately, after we reported this problem, the developer refused

[1]We are required to keep the company and the product anonymous.



Misconfiguration:
listener-threads 32
Symptom:
Crash after server startup with the only log message: "*Segmentation fault*".
Root Cause:
OpenLDAP only supports a hard-coded maximum of 16 listener threads.

The user manual does not mention this limit.

Developer's Response:
Refused to change the source code and the manual because *the setting is not valid*.

Figure 2.2. A real-world example from OpenLDAP. The LDAP server crashes when "`listener-threads`" is set to be larger than 16.

to take any action, such as changing the configuration design, editing the manual entry, or adding some code to check the value and print out explicit error messages. This was mainly due to the common attitude many developers have towards misconfigurations: "*It is not a bug, but an invalid setting.*"

Of course, not all developers are like this. Some developers have a more responsible attitude to handling misconfigurations. For example, after we reported the *misconfiguration vulnerabilities* (bad system reactions to configuration errors, such as crashes, hangs, incorrect results) and error-prone constraints to Squid (an open-source Web proxy and cache server), Squid developers fixed the reported problems immediately. Also, the large U.S. commercial company we worked with has been very cooperative, including allowing us to publish our evaluation results of their system.

2.3 Configuration Constraint Inference

This section describes the design and implementation of SPEX, a tool that automatically infers configuration constraints (i.e., rules that differentiate correct configurations from misconfigurations) from source code. In the next section, we will discuss how SPEX uses such constraints to expose misconfiguration vulnerabilities, and to detect error-prone configuration design and handling.

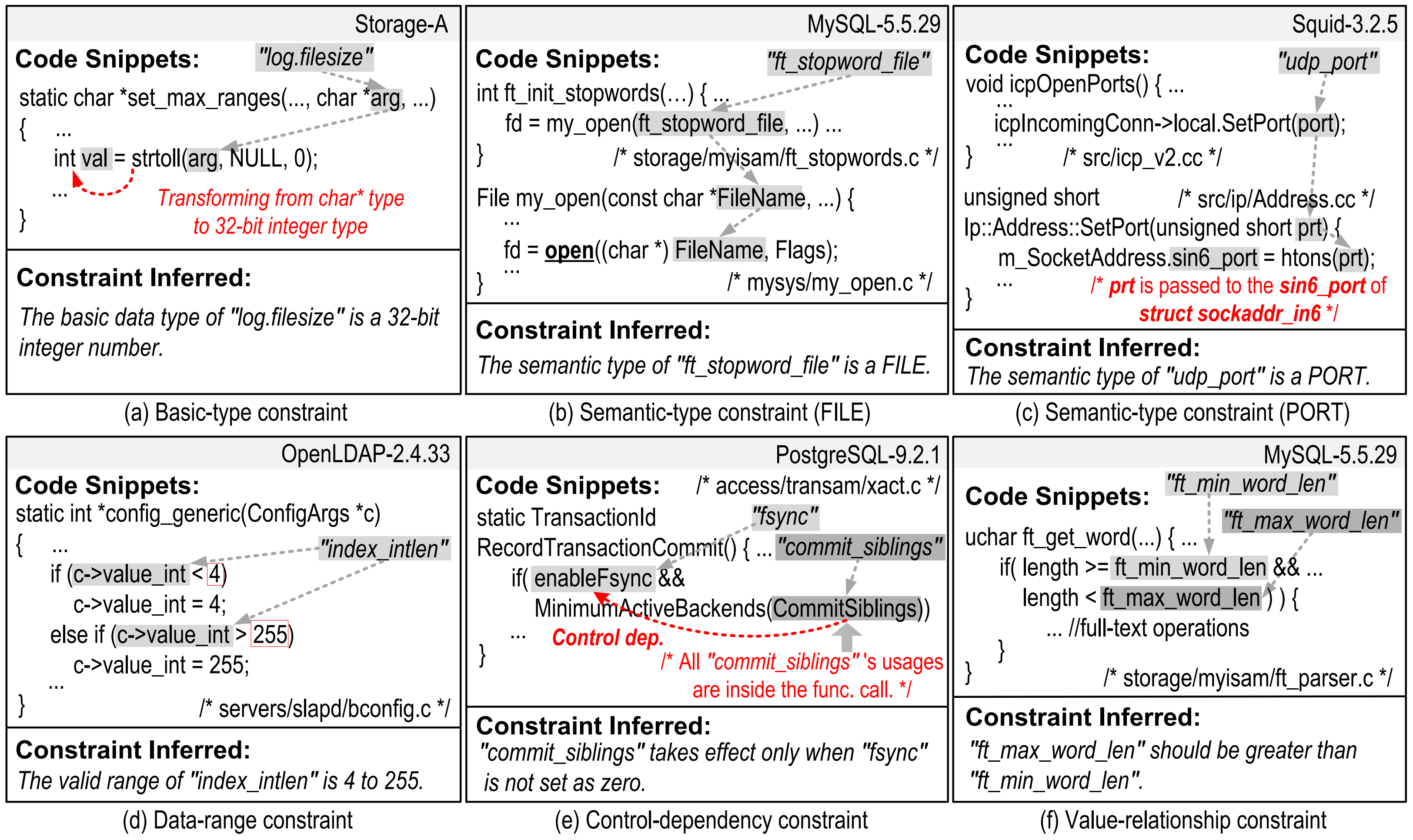
SPEX requires the target software's source code and simple annotations as a starting point to help identify and analyze configuration parameters in source code. This section first describes what kinds of configuration constraints SPEX can infer, and then discusses how to infer them. Finally, we discuss the limitations, in particular, what kinds of constraints cannot be inferred by SPEX.

2.3.1 What Constraints Can Be Inferred?

Many configuration requirements are reflected in the software's source code. Examples include data types, format, value ranges, multi-parameter dependencies, etc. Some of them can be automatically inferred via static code analysis by leveraging the properties of various operations, and system/library APIs when accessing configuration-related variables. Certainly, as we will discuss in §2.3.3, not all configuration constraints are reflected in source code or can be automatically inferred via static analysis. This chapter provides a first step in this direction. §2.5.1 shows promising results, with even a modest real-world impact on both commercial and open-source software.

SPEX analyzes source code and infers constraints that manifest through concrete, recognizable program patterns. These constraints can be classified into *attributes* and *correlations*. The former define the correct settings of a parameter, while the latter specify the correlations among multiple parameters. Figure 2.3 gives several concrete real-world examples of various kinds of configuration constraints SPEX infers. The following describes each kind in more detail and §2.3.2 will explain in detail how SPEX infers them, starting from how it identifies configuration variables in source code.

Data type. To set a configuration parameter correctly, operators first need to know the expected data type. We call such constraints *type constraints*. There are two categories of types for a configuration parameter: *basic types* and *semantic types*. The



Storage-A

Code Snippets: "log.filesize"

```
static char *set_max_ranges(..., char *arg, ...)
{   ...
    int val = strtoll(arg, NULL, 0);
    ...          Transforming from char* type
                   to 32-bit integer type
}
```

Constraint Inferred:
The basic data type of "log.filesize" is a 32-bit integer number.

(a) Basic-type constraint

MySQL-5.5.29

Code Snippets: "ft_stopword_file"

```
int ft_init_stopwords(...) { ...
   fd = my_open(ft_stopword_file, ...) ...
}                  /* storage/myisam/ft_stopwords.c */
File my_open(const char *FileName, ...) {
   ...
   fd = open((char *) FileName, Flags);
}  ...                     /* mysys/my_open.c */
```

Constraint Inferred:
The semantic type of "ft_stopword_file" is a FILE.

(b) Semantic-type constraint (FILE)

Squid-3.2.5

Code Snippets: "udp_port"

```
void icpOpenPorts() { ...
   ...
   icpIncomingConn->local.SetPort(port);
}  ...        /* src/icp_v2.cc */
unsigned short           /* src/ip/Address.cc */
Ip::Address::SetPort(unsigned short prt) {
    m_SocketAddress.sin6_port = htons(prt);
}   ...          /* prt is passed to the sin6_port of
                      struct sockaddr_in6 */
```

Constraint Inferred:
The semantic type of "udp_port" is a PORT.

(c) Semantic-type constraint (PORT)

OpenLDAP-2.4.33

Code Snippets:

```
static int *config_generic(ConfigArgs *c)
{   ...                         "index_intlen"
    if (c->value_int < 4)
       c->value_int = 4;
    else if (c->value_int > 255)
       c->value_int = 255;
    ...
}                         /* servers/slapd/bconfig.c */
```

Constraint Inferred:
The valid range of "index_intlen" is 4 to 255.

(d) Data-range constraint

PostgreSQL-9.2.1

Code Snippets: /* access/transam/xact.c */

```
static TransactionId              "fsync"
RecordTransactionCommit() { ... "commit_siblings"
    if( enableFsync &&
       MinimumActiveBackends(CommitSiblings))
    ...    Control dep.
}                /* All "commit_siblings"'s usages
                    are inside the func. call. */
```

Constraint Inferred:
"commit_siblings" takes effect only when "fsync" is not set as zero.

(e) Control-dependency constraint

MySQL-5.5.29

Code Snippets: "ft_min_word_len" "ft_max_word_len"

```
uchar ft_get_word(...) { ...
    if( length >= ft_min_word_len && ...
       length < ft_max_word_len ) ) {
       ... //full-text operations
    }
}                 /* storage/myisam/ft_parser.c */
```

Constraint Inferred:
"ft_max_word_len" should be greater than "ft_min_word_len".

(f) Value-relationship constraint

Figure 2.3. Real-world examples to illustrate the types of configuration constraints SPEX infers. The arrows show the data-flow, which motivates SPEX to do data-flow analysis. Configuration parameters are quoted in the figure, and the program variables that store the parameters are shaded. §2.3.2 explains how these constraints are inferred.

basic-type constraint specifies the parameter's value by low-level data representations, including integers, characters, boolean, floating-point numbers, strings, etc.

However, basic types alone may not be sufficient. For example, a `string` parameter may refer to either a file path or an IP address. Each such semantic type has its own specific requirements. For example, a file path has a specific path-like format and should represent a valid file in the file system. In addition to the "file path" and "IP address" types, there are many other types such as user name, port number, timeout, etc. In SPEX, we support the high-level semantic types of most standard libraries.

Figures 2.3(a), (b), and (c) show three real-world examples of type constraints inferred by SPEX. In the first example, via static code analysis, SPEX infers the parameter, "`log.filesize`", to be a 32-bit integer number. Figure 2.3(b) gives an example of the `file` type, and Figure 2.3(c) shows an example of the `port` type.

Value range. Configuration parameters may be further constrained by some acceptable ranges of valid values, such as minimum and maximum values, or a list of acceptable values as in the enumerative type. Figure 2.3(d) shows a value range constraint inferred by SPEX from OpenLDAP, in which, as the code indicates, the value of "`index_intlen`" needs to be between 4 and 255.

Control dependency. Multiple configuration parameters might have dependencies. Often times, the resolution to problems like, "*Why does my setting of parameter `P` not work?*" is simply "*Turn on parameter `Q`.*" When such dependencies are neither documented in the manual, nor pinpointed explicitly by log messages, it is difficult for operators to figure them out. Such constraints are typically manifested as *control dependencies* in source code. Formally, we define the control dependency of two parameters as $(\mathtt{P},\mathtt{C},\diamond) \mapsto \mathtt{Q}$ which means that the usage of parameter `Q` relies on the setting of parameter `P`, under the condition of $\texttt{P} \diamond \texttt{C}$, where $\diamond \in \{<, >, =, \neq, \geq, \leq\}$, and `C` is a constant value. Figure 2.3(e) shows an example from PostgreSQL, where "`commit_siblings`" takes effect only when "`fsync`" is non-zero.

Value relationship. Besides the control dependency between two parameters, the relationship of their values may also impose constraints. In Figure 2.3(f), the value of "`ft_max_word_len`" should be greater than that of "`ft_min_word_len`".

2.3.2 How to Infer Constraints?

To infer configuration constraints, SPEX first needs to identify *configuration variables* (program variables that initially store the values of configuration parameters loaded from configuration files) in source code. It then tracks the data-flow of each program variable corresponding to the configuration parameter, meanwhile recording any constraints that are discovered along the data-flow paths.

SPEX's analysis is implemented to be inter-procedural, context-sensitive, and field-sensitive. Inter-procedure is necessary, because configuration parameters are commonly passed through function calls. SPEX also needs to be field-sensitive, because configuration parameters could be stored in composite data types. SPEX is built on top of the LLVM compiler infrastructure [5]. As a design choice, we do not use symbolic execution for SPEX. Symbolic execution is able to explore all the possible code paths in the program, for the given input. However, it suffers from path explosion when applied to large systems such as Storage-A. Moreover, as shown in §2.3, SPEX looks for concrete code patterns on the data-flow paths of each configuration parameter, which does not fit the strength of symbolic execution.

SPEX scans the source code twice. In the first pass, it infers the data-flow path of each parameter, and looks for data-type and data-range constraints for each parameter.

To further infer constraints involving multiple parameters (i.e., control dependencies and value relationships), SPEX scans the code again, but this time only on the program slice containing the data-flow of each parameter.

Mapping Parameters to Variables

To start constraint inference, SPEX has to know the program variables that initially store the values of configuration parameters loaded from configuration files. Different software projects may have different conventions. We observe that developers often use well-defined *interfaces* to manage configurations. By examining 26 widely-used software projects (c.f., Table 2.1), we find that all but one of them map configuration parameters into program variables via one of the three interfaces: *structure*, *comparison*, and *getter*. Correspondingly, SPEX provides three template-based toolkits to extract the mapping information with minimal annotation efforts.

In structure-based mapping, data structures are used to directly map each configuration parameter to the corresponding variable in source code [c.f., Figure 2.4(a)], or to the parsing function [c.f., Figure 2.4(b)]. In the former case, developers only need to provide the structure variable's name and each specific field. For Figure 2.4(a), three lines of annotations are sufficient to extract the mapping information of 82 parameters in PostgreSQL. In the latter case, developers need to further annotate which parameter in the parsing function is the configuration variable [e.g., `arg` in Figure 2.4(b)]. We observed that structure-based mapping is the most common pattern in C/C++ programs.

Comparison-based mapping, as shown in Figure 2.4(c), uses string comparison functions (e.g., `strcasecmp`) to match parameters. It further assigns values to the variables in the branch blocks. SPEX recognizes standard string comparison functions. In this case, developers need to annotate the parsing function, and the initial input variables holding the parameter names and values.

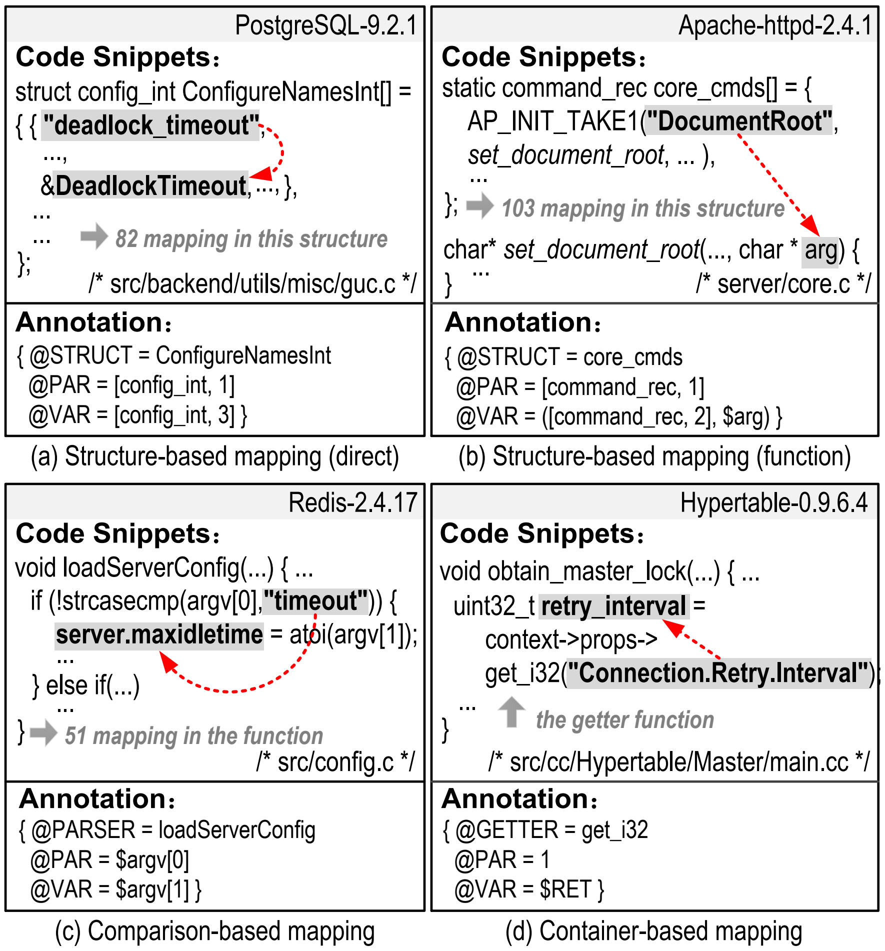
Table 2.1. Mapping from configuration parameters to program variables in 26 software projects. All of them fall into one of the three conventions, termed *structure*, *getter* and *comparison*, or their combinations.

Software	Desc.	Type	Software	Desc.	Type
Storage-A	storage	struct	Hypertable	database	getter
MySQL	database	struct	MongoDB	database	getter
PostgreSQL	database	struct	AOLServer	web server	getter
Apache httpd	web server	struct	MapReduce	computing	getter
lighttpd	web server	struct	YARN	scheduler	getter
Ngnix	web server	struct	HBase	database	getter
OpenSSH	SSH	struct	Accumulo	database	getter
Postfix	email	struct	HDFS	storage	getter
VSFTP	FTP	struct	Alluxio	storage	getter
Squid	proxy	comparison	Oozie	scheduler	getter
Redis	KV store	comparison	ZooKeeper	coordinator	getter
ntpd	NTP	comparison	OpenStack	cloud	getter
memcached	caching	comparison	Spark	computing	getter

Getter-based mapping, exemplified in Figure 2.4(d), stores all the configuration parameters in a central container, and uses common *getter* functions to retrieve the value. In such cases, developers need to annotate the getter functions (typically only a few). Getter-based mapping is the standard approach to manage configurations in both Java and Python programs, with the library support `java.util.Properties` and `configparser` respectively.

By asking developers to annotate the mapping interfaces rather than every mapping pair, the toolkits require limited amount of information from developers. In the evaluation, the number of annotations needed for most software is less than 10, as shown in Table 2.4. Note that the annotation only requires modest understanding of source code. The configuration-related code is usually modularized and can be found by simply searching parameter names in source code (e.g., using `grep`).

Starting from the annotations, the SPEX toolkits infer the mapping information in the form of key-value pairs: ("parameter name", variable name). For example, the key-value pair in Figure 2.4(a) is ("`deadlock_timeout`", `DeadlockTimeout`). In the



PostgreSQL-9.2.1

Code Snippets:

```
struct config_int ConfigureNamesInt[] =
{ { "deadlock_timeout",
    ...,
    &DeadlockTimeout, ..., },
  ...
  ...   82 mapping in this structure
};
          /* src/backend/utils/misc/guc.c */
```

Annotation:

```
{ @STRUCT = ConfigureNamesInt
  @PAR = [config_int, 1]
  @VAR = [config_int, 3] }
```

(a) Structure-based mapping (direct)

Apache-httpd-2.4.1

Code Snippets:

```
static command_rec core_cmds[] = {
   AP_INIT_TAKE1("DocumentRoot",
   set_document_root, ... ),
   ...
}; 103 mapping in this structure
char* set_document_root(..., char * arg) {
}  ...                      /* server/core.c */
```

Annotation:

```
{ @STRUCT = core_cmds
  @PAR = [command_rec, 1]
  @VAR = ([command_rec, 2], $arg) }
```

(b) Structure-based mapping (function)

Redis-2.4.17

Code Snippets:

```
void loadServerConfig(...) { ...
  if (!strcasecmp(argv[0],"timeout")) {
     server.maxidletime = atoi(argv[1]);
     ...
  } else if(...)
     ...
} 51 mapping in this function
                      /* src/config.c */
```

Annotation:

```
{ @PARSER = loadServerConfig
  @PAR = $argv[0]
  @VAR = $argv[1] }
```

(c) Comparison-based mapping

Hypertable-0.9.6.4

Code Snippets:

```
void obtain_master_lock(...) { ...
  uint32_t retry_interval =
      context->props->
      get_i32("Connection.Retry.Interval");
  ...     the getter function
}
     /* src/cc/Hypertable/Master/main.cc */
```

Annotation:

```
{ @GETTER = get_i32
  @PAR = 1
  @VAR = $RET }
```

(d) Container-based mapping

Figure 2.4. Examples of three mapping conventions and the corresponding annotations to get the mapping information

remainder of this section, we refer to the variables storing the configuration parameters' values as "parameters," to simplify our description.

Since SPEX was published [179], this approach has become a common practice for code-base configuration analysis to map configuration parameters to the corresponding program variables that store their values [26, 44, 79, 123, 124, 179, 191, 192]. More sophisticated approaches that leverage machine learning [197] and string analysis [43] have also been proposed, which can be directly plugged into SPEX.

Data Type Inference

Basic type. SPEX infers each configuration parameter's basic type from its type information in the source code. On the data-flow path of a parameter, its type might be casted (changed) multiple times. In such cases, SPEX records the type after the first casting as the basic type, because it is common for a parameter to be first stored as a string (e.g., a `char` array), before being transformed into its real type. Figure 2.3(a) shows an example from the commercial software Storage-A, in which the configuration parameter is converted from a string to a 32-bit integer. Thus, the basic type constraint of "`log.filesize`" is inferred as 32-bit integer.

Semantic type. SPEX infers semantic-type constraints by searching the following patterns along a parameter's entire data-flow path: (1) the parameter is passed to a known function call (e.g., system- and library-call) or a known data structure; or (2) the parameter is compared with or is assigned with the return value of a known function call (e.g., the return value of the `time` syscall). Figure 2.3(b) shows an example from MySQL of the first pattern. In this example, SPEX infers the semantic type of "`ft_stopword_file`" to be a file path because it is used in the `open` syscall. Note that SPEX searches such patterns along the entire data-flow path, even after the parameter is modified, because the modification seldom affects the semantic type. For example, a file path after canonicalization is still used as a file path.

SPEX supports the standard library APIs and data types. We also allow developers to import their own library APIs and data types by pointing to their header files. For example, for the commercial storage software used in the evaluation, we also imported its proprietary library APIs. For constraint inference, the library APIs included in `.h` files are enough, but for misconfiguration injection described in §2.4.1, SPEX needs developers to provide types of configuration errors to inject, for customized data types. Note that they do not need to provide such information for types defined in standard libraries (in the evaluation, customization is used only for the commercial storage system).

Data Range Inference

SPEX infers range constraints when the parameter is compared with constant values in conditional branches. SPEX infers two types of ranges: numeric and enumerative. For numeric comparison, SPEX treats the constant numbers as *thresholds* of the data range. Enumerative ranges are inferred if the parameter is used in `switch` statements or `if..else if..else` logic.

For each range inferred, SPEX further decides whether the range is valid or not by analyzing the program behavior within the corresponding branch blocks. The reason for inferring such information is to guide misconfiguration injection to expose bad system reactions. If in the branch block, the program exits, aborts, returns error code, or resets the parameter, SPEX treats the range as invalid. Otherwise, it is valid. Figure 2.3(d) shows an example of range inference from OpenLDAP, in which the range of "index_intlen" is divided into $(-\infty, 4)$, $[4, 255]$, and $(255, +\infty)$. Both $(-\infty, 4)$ and $(255, +\infty)$ are invalid because the parameters are reset in those ranges. The `default` in a `switch` statement or the last `else` in `if..else if..else` logic is also treated as invalid. Please note, since such information is used to guide misconfiguration injection (c.f., §2.4.1), some false positives are not a major concern.

As a good practice, range constraints should be explicitly documented, but this is not always the case. As shown in Figure 2.3(d), OpenLDAP limits index lengths within $[4, 255]$. However, this constraint is not documented. If operators set out-of-range values, the system misbehaves with no warning, leaving operators clueless.

Control Dependency Inference

To infer control dependencies, SPEX starts from the usage statements of a parameter `Q`, and looks for conditional branches that dominate these statements in a bottom-up manner. If the condition involves the variable that is part of the data-flow of another parameter `P`, SPEX records a control dependency between `P` and `Q` in the form of $(\texttt{P},\texttt{C},\diamond) \mapsto \texttt{Q}$. Figure 2.3(e) gives an example of a control dependency from PostgreSQL. Starting from the usage statement of “`commit_siblings`” inside a function call (omitted in the figure), SPEX goes backwards to check the conditions that allow the execution of this usage, and infers (“`fsync`”$, 0, \neq) \mapsto$ “`commit_siblings`” as the control dependency. Note that passing a parameter to a function and modifying its value are not considered as *usage*, because they do not change program behavior [144]. To be considered as usage statements, they have to be used in branches, arithmetic operations, or as system-/library-call arguments.

However, if we blindly treated every such occurrence of control dependencies as constraints, there would be many false ones. For example, VSFTP has three settings: “`listen`” (for ipv4), “`listen_ipv6`”, and “`listen_port`”. “`listen_port`” is used after the check of “`listen`” and the check of “`listen_ipv6`”. If we blindly generated two configuration constraints: (“`listen`”$, 1, =) \mapsto$ “`listen_port`” and (“`listen_ipv6`”$, 1, =) \mapsto$ “`listen_port`”, both would be too strict. To handle this problem, SPEX aggregates all the inferred control dependencies for each configuration parameter from *all* control-flow paths, and calculates the MAY-belief confidence of each dependency in a way similar to [48]. Only if the confidence exceeds a predefined threshold (currently set to 0.75), the control dependency will be reported. In the above example, each dependency will have a confidence of 0.5, not exceeding the threshold. Therefore, both of them are filtered out.

Value Relationship Inference

Similar to control-dependency inference, the value relationship also involves multiple parameters. SPEX looks for comparison statements in parameters' usage. If two variables from different parameters' data-flow paths are compared with each other, SPEX infers the value relationship of the two parameters in the form of $P \diamond Q$. In addition, the value relationship is transitive, which means it can be transited through intermediate variables. Figure 2.3(f) gives such an example from MySQL that the min-max relation is transited by a local variable. In the current prototype of SPEX, we only check one intermediate variable for transitivity, which is fast and captures common cases. SPEX further tries to decide whether the inferred relationship indicates a valid setting or not, in a manner similar to that in range-constraint inference.

2.3.3 Discussion and Limitation

No tool is perfect, and SPEX is no exception. SPEX cannot infer all configuration constraints and it also has false positives, even though the evaluation with commercial and open-source software has shown good results.

Currently, the constraint inference of SPEX is limited within the scope of a single program's source code. However, when we study real-world misconfiguration issues (c.f., §2.5.2), we find that cross-software configuration correlations also account for a considerable number of misconfiguration cases. Inferring these constraints requires new techniques to consider the software stacks as a whole, which remains future work.

Even within a single program, SPEX does not infer all constraints. Some constraints are program-specific, without common, concrete program patterns. For example, it is hard for SPEX to understand the complicated string manipulation logics used in parsing certain parameters (e.g., nesting and semi-structured rules), which might appear in software providing services of networking and access controls (e.g., Bind9, Netfilter).

Moreover, SPEX cannot infer all the possible semantics of parameters.

The constraints inferred by SPEX are basic, and cannot capture certain complicated constraints (e.g., dependencies involving complicated compositions of boolean or arithmetic operations). On the other hand, according to our inspection, systems seldom have these complicated constraints on their configurations, possibly because most operators cannot handle such complexity.

Not every constraint inferred by SPEX is a true constraint. §2.5.3 provides the evaluation results for false positives. SPEX's inference accuracy is above 90% for most evaluated software. To further improve accuracy, we would need developers to manually examine each constraint and prune out the 10% false ones.

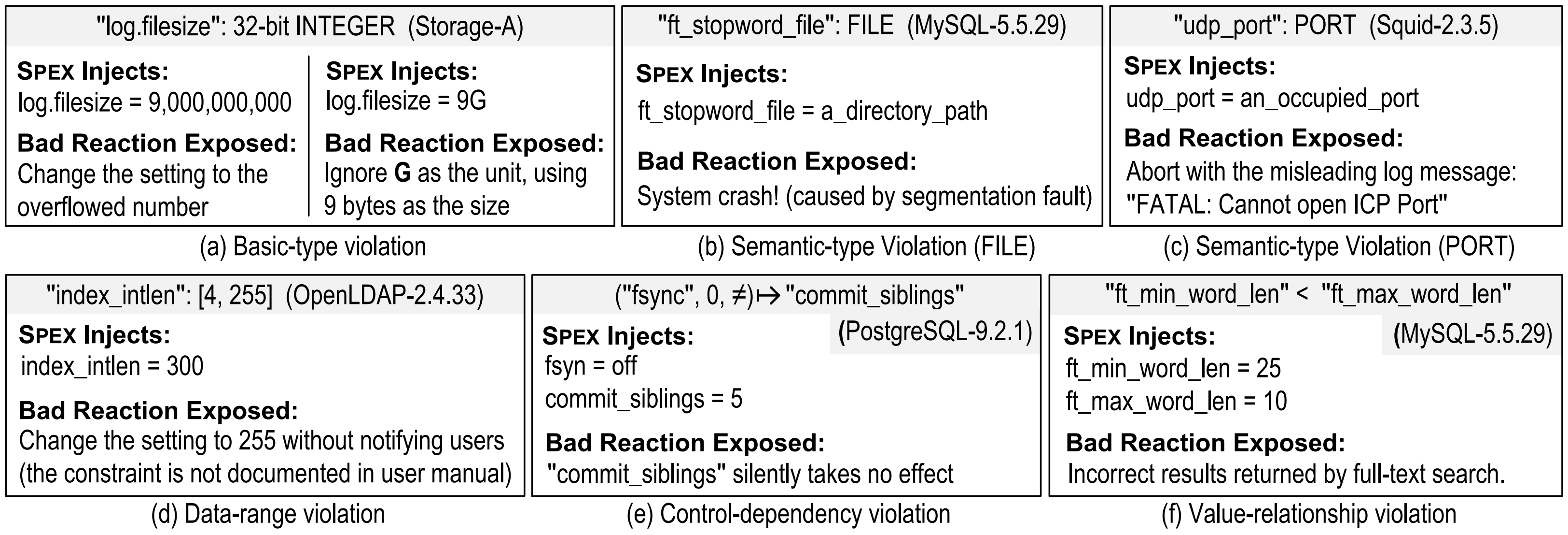
The analysis of SPEX works on LLVM's intermediate code representation (IR) which is a generic assembly language in the static single assignment (SSA) form. Thus, SPEX is applicable to software programs in programming languages that can be compiled into LLVM IR. In the implementation, we use Clang as the front-end tool to compile C/C++ source code into LLVM IR.

2.4 Use Cases of Configuration Constraints

We build two use cases to demonstrate that the constraints inferred by SPEX provide useful information for developers to help systems anticipate and defend against misconfigurations in terms of both configuration design and implementation.

2.4.1 Harden Systems against Configuration Errors

Given the configuration constraints inferred by SPEX, we take one step further. We build a misconfiguration injection-based testing tool called SPEX-INJ, to expose misconfiguration vulnerabilities. SPEX-INJ automatically generates misconfigurations by violating the constraints inferred by SPEX. Then, it injects these misconfigurations



"log.filesize": 32-bit INTEGER (Storage-A)

SPEX Injects:
log.filesize = 9,000,000,000
Bad Reaction Exposed:
Change the setting to the overflowed number

SPEX Injects:
log.filesize = 9G
Bad Reaction Exposed:
Ignore **G** as the unit, using 9 bytes as the size

(a) Basic-type violation

"ft_stopword_file": FILE (MySQL-5.5.29)

SPEX Injects:
ft_stopword_file = a_directory_path
Bad Reaction Exposed:
System crash! (caused by segmentation fault)

(b) Semantic-type Violation (FILE)

"udp_port": PORT (Squid-2.3.5)

SPEX Injects:
udp_port = an_occupied_port
Bad Reaction Exposed:
Abort with the misleading log message:
"FATAL: Cannot open ICP Port"

(c) Semantic-type Violation (PORT)

"index_intlen": [4, 255] (OpenLDAP-2.4.33)

SPEX Injects:
index_intlen = 300
Bad Reaction Exposed:
Change the setting to 255 without notifying users
(the constraint is not documented in user manual)

(d) Data-range violation

("fsync", 0, ≠)↦"commit_siblings" (PostgreSQL-9.2.1)

SPEX Injects:
fsyn = off
commit_siblings = 5
Bad Reaction Exposed:
"commit_siblings" silently takes no effect

(e) Control-dependency violation

"ft_min_word_len" < "ft_max_word_len" (MySQL-5.5.29)

SPEX Injects:
ft_min_word_len = 25
ft_max_word_len = 10
Bad Reaction Exposed:
Incorrect results returned by full-text search.

(f) Value-relationship violation

Figure 2.5. Real-world examples to illustrate the configuration error generation of SPEX-INJ (based on the rules in Table 2.2), and the exposed misconfiguration vulnerabilities (bad system reactions). How the constraints are inferred from these examples is shown in Figure 2.3. All the vulnerabilities are detected by SPEX-INJ in the latest versions of the evaluated systems.

Table 2.2. SPEX-INJ's generation of misconfigurations for different types of constraints inferred by SPEX

Constraint	**Generation Rules**
Basic type	Generate parameter values with invalid basic types
Semantic type	Generate invalid values specific to different semantic types
Range	Generate out-of-range values
Control dependency	Generate $(\mathtt{P}\ \overline{\diamond}\ \mathtt{C}) \land \mathtt{Q}$ for $(\mathtt{P},\mathtt{C},\diamond) \mapsto \mathtt{Q}$
Value relationship	Generate invalid value relationships

to the configuration settings, and tests how the system reacts. If the system does not react well (e.g., crashes, hangs, and dysfunctions), SPEX-INJ reports the bad reactions to the developers. By fixing these vulnerabilities, e.g., adding checks and log messages to detect and pinpoint the error, developers can harden systems against potential misconfigurations introduced in the field, and allow operators to quickly find their configuration errors so as to fix the errors by themselves.

Misconfiguration generation and injection. Table 2.2 summarizes how SPEX-INJ generates configuration errors by intentionally violating the inferred constraints. Each misconfiguration includes one or several erroneous parameter values that violate a specific constraint. SPEX-INJ may generate several misconfigurations in various aspects for a parameter: violating the constraints of its data type, its data range, its dependencies and correlations with other parameters. Every generation rule is implemented as a plug-in. It is easy to be extended to customize generation rules. Figure 2.5 lists several real-world examples for each rule along with the exposed vulnerabilities.

SPEX-INJ injects the misconfigurations by replacing the default parameter values with the generated erroneous values in configuration files. We use the configuration file parser in ConfErr [83] to parse a template configuration file into an abstract representation (AR), and transform the modified AR with error injected to a usable configuration file for testing; other configuration file parsing tools such as Augeas can also be used.

Categories of misconfiguration vulnerabilities. When a misconfiguration occurs, the system should pinpoint either the misconfigured parameter's name/value or its location information (e.g., line numbers in the file). Otherwise, SPEX-INJ considers the system reaction as a *misconfiguration vulnerability*.

Table 2.3 categorizes different types of misconfiguration vulnerabilities. The first category, system crashes and hangs, is considered as severe vulnerabilities, especially for server applications where availability is crucial. Such symptoms would mislead operators and support engineers to suspect it as a software bug. The second category, early termination without pinpointing message, is also undesirable. In this case, the system terminates itself but does not give useful feedback for operators to fix the problems by themselves. Similarly, function failures without pinpointing error messages can also confuse operators, as shown in the MySQL example in Figure 2.5(f). As for the last two categories, it can still be unacceptable (maybe less severe) to silently violate or ignore the operator's intention, which might cause the operator's confusion or sophisticated problems (e.g., performance issues, feature not activated), as shown in Figure 2.5(a) and (d). Note that we do not consider performance issues caused by misconfigurations, mainly due to the difficulties in objectively judging if the performance is acceptable. Unless the performance degradation affects the system usability (belonging to "hang"), we consider it acceptable as long as the functionality is correct.

Testing and analysis. SPEX-INJ leverages each software's own test infrastructure including test cases and oracles for accepting/rejecting test results. For each generated configuration file (containing one misconfiguration), SPEX-INJ first launches the target system. If the system successfully starts, SPEX-INJ will further apply existing functional test cases one by one and monitor the system status and output. During testing, SPEX-INJ records all the system and console logs. If the test results fail to pass the

Table 2.3. The categories of bad system reactions

Reaction	Description
Crash/Hang	The system crashes or hangs˙
Early termination	The system exits abnormally without pinpointing the injected configuration error.
Functional failure	The system fails functional testing without pinpointing the injected configuration error.
Silent violation	The system changes the input configurations to different values without notifying the operators.
Silent ignorance	The system ignores the input configurations (mainly due to control-dependency violation).

test oracles, SPEX-INJ checks the logs to see whether the system pinpoints the misconfiguration. If not, it generates an error report for the developers.

The error report (i.e., the output of SPEX-INJ) contains the constraint, the injected error, and the failed test cases, associated with all the log messages. Therefore, the developers can know what misconfigurations caused what problems. SPEX-INJ reports silent violation/ignorance if the system does not pinpoint errors but passes testing.

This process can be slow, as $N \times T$, where N is the number of misconfigurations SPEX-INJ generates, and T is the time to run all input test cases once. To shorten the time, we apply two optimizations. First, for each misconfiguration, SPEX-INJ stops immediately after the first failed test case. Second, we sort the running time of each test case, and run the shortest test case first. By using these optimizations, the testing time of SPEX-INJ on the evaluated software is under 10 hours. Note that this is a one-time cost because SPEX-INJ can be made incrementally. It only needs to retest those constraints affected by code modification during each revision.

2.4.2 Detect Error-Prone Design and Handling

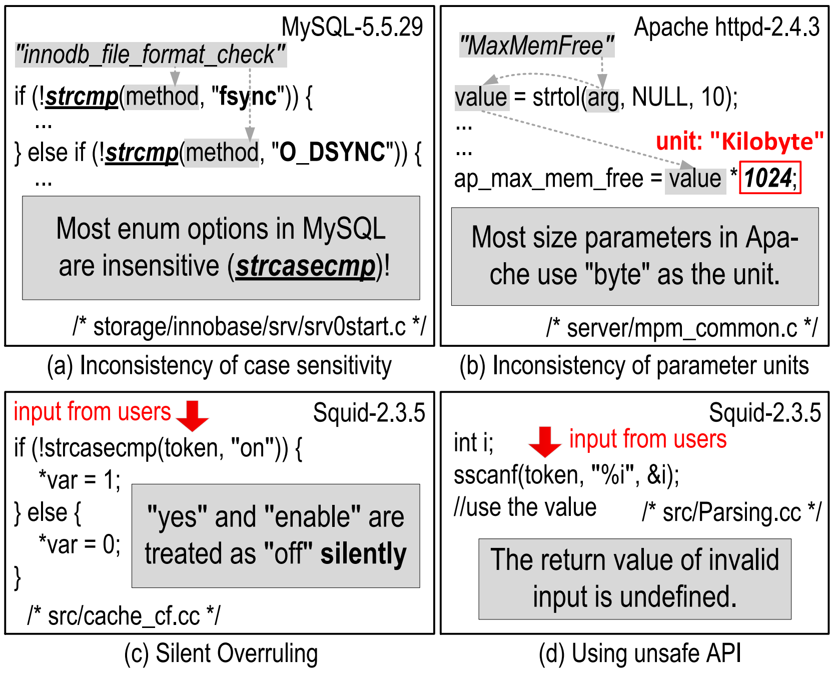
Configuration settings which are expected to be performed by operators, should be intuitive and less prone to errors. Carefully-designed configuration constraints can prevent operators confusion and mistakes. More specifically, since configuration setting is also one type of software interface exposed to the operators, it should follow the interface design principles [98, 111].

We expect the configuration design to be (1) *consistent* in constraints of different parameters, (2) *explicit* to operators when changing their configuration settings, and (3) *complete* in documenting the requirements of parameters (i.e., their constraints). In this section, we show how to leverage the constraints to detect error-prone configuration design and handling that break the three principles.

Design inconsistency. Consistency is a primary interface design principle to prevent operator mistakes. The inferred constraints provide opportunities for detecting two types of configuration inconsistency: (1) case sensitivity, and (2) unit granularity. Such inconsistency is error-prone because operators are likely confused by the contradictory requirements for parameters of same types.

Figures 2.6(a) and (b) show two real-world examples of the two types of inconsistency. In Figure 2.6(a), different from most string case-insensitive configuration parameters in MySQL, the values of parameter "`innodb_file_format_check`" are case sensitive. In Figure 2.6(b), different from the other *size* parameters in Apache that use *Bytes* as the unit, "`MaxMemFree`" uses *KBytes* as the unit. Therefore, operators can easily make mistakes here due to the inconsistency. As shown in § 2.5.1, we find that more than half of the evaluated systems have these two kinds of inconsistency.

The inconsistency is detected based on SPEX's inference of semantic-type constraints (remember that SPEX records the API calls that use the parameters). The case sensitivity is inferred by identifying string comparison functions. If the parameter is used in comparison functions like `strcasecmp`, it is case insensitive. Otherwise it is sensitive when used in functions like `strcmp`. Similarly, the unit information is in-


MySQL-5.5.29
Most enum options in MySQL are insensitive (strcasecmp)!
Apache httpd-2.4.3
unit: "Kilobyte"
Most size parameters in Apache use "byte" as the unit.
input from users
Squid-2.3.5
"yes" and "enable" are treated as "off" silently
Squid-2.3.5
input from users
The return value of invalid input is undefined.

```
"innodb_file_format_check"
if (!strcmp(method, "fsync")) {
  ...
} else if (!strcmp(method, "O_DSYNC")) {
  ...
/* storage/innobase/srv/srv0start.c */
```

(a) Inconsistency of case sensitivity

```
"MaxMemFree"
value = strtol(arg, NULL, 10);
...
...
ap_max_mem_free = value * 1024;
/* server/mpm_common.c */
```

(b) Inconsistency of parameter units

```
if (!strcasecmp(token, "on")) {
    *var = 1;
} else {
    *var = 0;
}
/* src/cache_cf.cc */
```

(c) Silent Overruling

```
int i;
sscanf(token, "%i", &i);
//use the value
/* src/Parsing.cc */
```

(d) Using unsafe API

Figure 2.6. Real-world examples of error-prone configuration design and handling

ferred according to the API's unit. For example, parameters used in `sleep` have the unit second, while parameters used in `usleep` are of unit microsecond. We also consider the transformation of the parameter, along its data-flow path before it falls into the API call, as shown in Figure 2.6(b).

Silent overruling. Silent overruling refers to the case that the system changes an unacceptable setting into the default value, without notifying the operator. This may cause silent violation of operator intention, as one type of the misconfiguration vulnerabilities. As shown in Figure 2.6(c), Squid silently treats any boolean parameter as "`off`" as long as it is not set to "`on`", even if it is set to "`yes`" or "`enable`". Such design can easily confuse operators because the system behavior would not match their expectation.

To detect silent overruling, for data range constraints detected in `switch` or `if...else if...else` logic, if the parameter's value is silently overwritten in the `default` case or the `else` block, we flag it as silent overruling. In Squid and Apache, we detected many silent overruling cases that affect 74 parameters. All of them have been fixed by developers after we reported them.

We do not consider static initialization of configuration parameters as silent overruling. It is mainly used to assign default values that would be overwritten by operator settings. Thus, it is not relevant to operator configuration.

Unsafe APIs. Using unsafe APIs in configuration handling can also create confusing behavior. For example, unsafe string-to-number transformation APIs, including `atoi`, `sscanf` and `sprintf` are vulnerable to erroneous operator inputs. Taking `atoi` as an example, there is no way to check unexpected characters (e.g., `atoi(1O0)` returns 1) and overflow issues (e.g., `atoi(INT_MAX)` returns -1). These APIs are handy in controlled contexts but should be avoided in configuration parsing since operator inputs may not be trustworthy and can easily be misspelled. Instead, a good practice is to use safe APIs such as `strtol` and check errors through `errno` and end pointers. Most bug detection tools do not report these vulnerabilities because they do not know whether a variable stores a configuration value. SPEX can detect them exactly, because it is starting from the value of configuration parameters. Our evaluation shows that many systems use unsafe transformation APIs, affecting large numbers of configuration parameters, as exemplified in Figure 2.6(d).

Undocumented constraints. The inferred constraints are also useful for developers to check whether some constraints are documented in any form, including operator manuals, error messages, or even accurate parameter naming. Our evaluation shows that some configuration constraints have never been documented in any form. As the consequence, operators can easily make mistakes with them.

Table 2.4. Evaluated software systems. “LoA” represents *lines of annotations*.

Software	**Proprietary**	**LoC**	**#Parameter**	**LoA**
Storage-A	Commercial	confidential		5
Apache	Open source	148K	103	4
MySQL	Open source	1.2M	272	29
PostgreSQL	Open source	757K	231	7
OpenLDAP	Open source	292K	86	4
VSFTP	Open source	16K	124	5
Squid	Open source	180K	335	2

2.5 Evaluation

We evaluate the effectiveness of the tools using one commercial system and six open-source systems as listed in Table 2.4. The commercial system, Storage-A, is from a major storage vendor in the U.S. It is a distributed storage operating system used for managing network attached storage devices. Storage-A serves storage over networks using both file-based protocols, including NFS, CIFS, FTP, HTTP, and block-based protocols, including FC, FCoE, iSCSI. The system provides operators with a large number of configuration parameters. The open-source systems are mature, widely-used server applications with considerable numbers of configuration parameters. The test cases we use to drive SPEX-INJ are from the test suites shipped with the software or provided by the developers. To collect related warning and error log messages, we set sufficient logging verbosity.

Table 2.4 shows the numbers of annotations we added in each software so that SPEX can use them as the starting points to identify and analyze configuration-related variables. As shown in the table, the annotation efforts in terms of lines is acceptable.

2.5.1 Overall Results

We first present the end results of SPEX—the misconfiguration vulnerabilities (bad system reactions) exposed by SPEX-INJ, as well as error-prone configuration de-

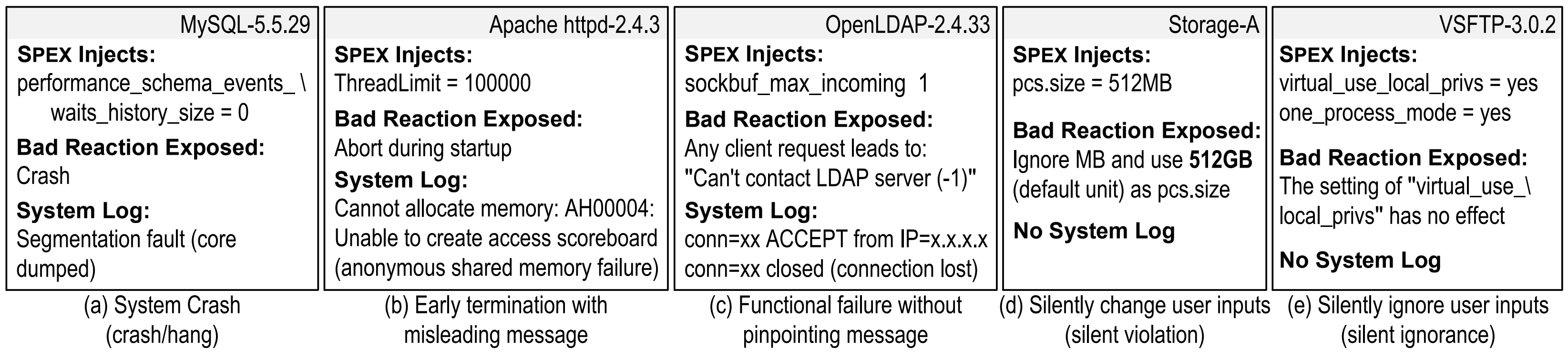
Table 2.5. The number of exposed misconfiguration vulnerabilities and the corresponding source-code locations. A patch for one source-code location might fix multiple vulnerabilities. The numbers in "()" are the numbers of confirmed or fixed cases by the developers after we reported them. The cases that have not been confirmed are discussed in §2.6.1

Software	**Crash/ Hang**	**Early terminat.**	**Functional failure**	**Silent violation**	**Silent ignor.**	**Total**
Storage-A	0 (0)	0 (0)	7 (5)	74 (72)	83 (0)	**164 (77)**
Apache	5 (2)	4 (3)	9 (3)	29 (2)	5 (1)	**52 (11)**
MySQL	5 (5)	10 (3)	12 (4)	71 (70)	16 (0)	**114 (82)**
PostgreSQL	1 (0)	10 (1)	2 (0)	1 (0)	35 (2)	**49 (3)**
OpenLDAP	1 (0)	3 (0)	6 (0)	7 (0)	0 (0)	**17 (0)**
VSFTP	12 (12)	5 (0)	18 (0)	23 (0)	68 (0)	**126 (12)**
Squid	2 (2)	3 (2)	29 (1)	173 (173)	14 (1)	**221 (179)**
Total	26 (21)	35 (9)	83 (13)	378 (317)	221 (4)	**743 (364)**

(a) Misconfiguration vulnerabilities (bad system reactions)

Software	**Code location**
Storage-A	119 (34)
Apache	52 (1)
MySQL	46 (16)
PostgreSQL	44 (3)
OpenLDAP	17 (0)
VSFTP	107 (12)
Squid	62 (21)
Total	**448 (97)**

(b) Corresp. code locations



MySQL-5.5.29

SPEX Injects:
```
performance_schema_events_\
    waits_history_size = 0
```
Bad Reaction Exposed:
Crash

System Log:
```
Segmentation fault (core dumped)
```
(a) System Crash (crash/hang)

Apache httpd-2.4.3

SPEX Injects:
```
ThreadLimit = 100000
```
Bad Reaction Exposed:
Abort during startup

System Log:
```
Cannot allocate memory: AH00004:
Unable to create access scoreboard
(anonymous shared memory failure)
```
(b) Early termination with misleading message

OpenLDAP-2.4.33

SPEX Injects:
```
sockbuf_max_incoming  1
```
Bad Reaction Exposed:
Any client request leads to:
"Can't contact LDAP server (-1)"

System Log:
```
conn=xx ACCEPT from IP=x.x.x.x
conn=xx closed (connection lost)
```
(c) Functional failure without pinpointing message

Storage-A

SPEX Injects:
```
pcs.size = 512MB
```
Bad Reaction Exposed:
Ignore MB and use **512GB** (default unit) as pcs.size

No System Log

(d) Silently change user inputs (silent violation)

VSFTP-3.0.2

SPEX Injects:
```
virtual_use_local_privs = yes
one_process_mode = yes
```
Bad Reaction Exposed:
The setting of "virtual_use_\local_privs" has no effect

No System Log

(e) Silently ignore user inputs (silent ignorance)

Figure 2.7. Examples of different types of misconfiguration vulnerabilities (categorized in Table 2.3) exposed by SPEX-INJ.

sign and handling. We then show the intermediate results, the configuration constraints inferred by SPEX in §2.5.3.

Misconfiguration vulnerabilities. Table 2.5(a) shows the number of misconfiguration vulnerabilities (bad system reactions) exposed in the latest versions of the evaluated systems. SPEX-INJ exposes a total of 743 vulnerabilities (they are true vulnerabilities verified by us). To this day, 364 of them have been confirmed or fixed by the developers. The vulnerabilities exposed by SPEX-INJ are of various kinds in all the evaluated systems. Most notably, all the open-source systems experienced bad reactions such as crashes, hangs, and early terminations under some misconfigurations. In addition, silent violation and ignorance are more prevalent compared with terminations and failures. This once again reflects that developers pay less attention to defending against misconfigurations, as long as they do not affect the system's own execution. Figure 2.7 gives five additional examples for each type of vulnerabilities exposed by SPEX-INJ.

Since one source-code location could affect the constraints of several configuration parameters, Table 2.5(b) further shows the number of unique code locations that cause these vulnerabilities. The 743 vulnerabilities are caused by 448 locations in source code, and the 364 confirmed bad reactions can be fixed by 97 code patches.

Error-prone configuration design and handing. Table 2.6 shows the distribution of the case-sensitivity requirements for string parameters in each system. We can see that more than half of the systems have inconsistent case-sensitivity requirements. The inconsistent requirements of 80 parameters in Apache, MySQL, and Squid have been confirmed and fixed after we reported them.

Table 2.7 shows the unit requirements for *size* and *time* parameters. More than half of the systems have inconsistent *size* and *time* units. For example, in Storage-A,

Table 2.6. Case-sensitivity requirements of different configuration parameters

Software	Case sensitivity		Developers' fixes
	Sensitive	**Insensitive**	
Storage-A	**32 (7.1%)**	**453 (92.9%)**	being investigated
Apache	**3 (11.5%)**	**26 (88.5%)**	all sens.→insens.
MySQL	**1 (1.7%)**	**58 (98.3%)**	all sens.→insens.
PostgreSQL	0 (0.0%)	92 (100.0%)	N/A
OpenLDAP	0 (0.0%)	9 (100.0%)	N/A
VSFTP	0 (0.0%)	73 (100.0%)	N/A
Squid	**85 (52.8%)**	**76 (47.2%)**	all insens.→sens.

Table 2.7. Different units of size- and time-related configuration parameters

Software	Size				Time				
	B	KB	MB	GB	μs	ms	s	m	h
Storage-A	**20**	**1**	**1**	**1**	**2**	**10**	**53**	**12**	**4**
Apache	**20**	**1**	0	0	0	**1**	**26**	0	0
MySQL	**29**	0	0	0	**2**	**2**	**13**	0	0
PostgreSQL	**1**	**3**	0	0	**1**	**12**	**9**	**1**	0
OpenLDAP	**2**	0	0	0	0	0	**3**	0	0
VSFTP	**1**	0	0	0	0	0	**6**	0	0
Squid	**18**	**2**	0	0	**1**	**6**	**33**	0	0

20 *size* parameters use Bytes as their units except three parameters, each of which uses different unit size, namely KBytes, MBytes, and GBytes. Storage-A mitigates the inconsistency via naming: including the unit information in parameter names (c.f., §2.6.2). However, none of the open-source systems makes such efforts, so the inconsistencies may confuse operators and cause mistakes.

Table 2.8 shows other types of error-prone constraints. SPEX detects 74 parameters with silent overruling in Apache and Squid, all of which were fixed by the developers after we reported them. In addition, more than half of the systems use unsafe transformation APIs for large numbers of parameters. Moreover, a number of inferred constraints are not documented in any form.

However, it might be arguable whether the cases in Table 2.7 and 2.8 are confusing and error-prone to operators. To be conservative, we did not report them to the

Table 2.8. The other types of error-prone configuration design and handling

Software	**Silent over-ruling**	**Unsafe trans-form.**	**Undoc. Constraints**		
			Data range	Ctrl dep.	Val. rel.
Storage-A	0	28	2	0	2
Apache	1	27	0	1	0
MySQL	0	0	4	3	1
PostgreSQL	0	0	3	3	2
OpenLDAP	0	0	2	0	0
VSFTP	0	20	3	47	1
Squid	73	115	3	4	4

developers. For the same reason, we did not include them in the results presented in the abstract and introduction section.

2.5.2 Benefits to Real-World Configuration Problems

It is hard to predict the benefits of SPEX in avoiding future misconfiguration reports and in reducing misconfiguration diagnosis time. To provide some estimation of the end benefits, we have to leverage past misconfiguration cases committed by real operators and evaluate how many customer reports could have been avoided, if the tools had been used. Note that the results in this section are from the perspective of system vendors. We do not consider the customers downtime and frustration.

We study real-world historical misconfiguration cases from four systems: the commercial system (Storage-A), Apache, MySQL, and OpenLDAP. For Storage-A, we randomly sampled 246 parameter misconfiguration cases from the company's customer issue database. For open-source applications, we randomly collected 177 parameter misconfigurations from official forums, mailing lists, and ServerFault.com (a popular system administration forum). The data have been presented in [185].

As shown in Table 2.9, 24%–38% of the misconfiguration cases could have been potentially avoided if SPEX had been used to improve the configuration design and

Table 2.9. Real-world misconfiguration cases that can be potentially avoided among all sampled historic cases

Software	Parameter misconfig.	Bad reactions that could be potentially avoided by SPEX
Storage-A	246	68 (27.6%)
Apache	50	19 (38.0%)
MySQL	47	14 (29.8%)
OpenLDAP	49	12 (24.5%)

Table 2.10. The breakdown of misconfigurations that cannot benefit from SPEX

Software	Inference incapability		Conform to constraints	Good reaction
	Single-SW	Cross-SW		
Storage-A	19 (7.7%)	51 (20.7%)	76 (30.9%)	32 (13.0%)
Apache	5 (10.0%)	12 (24.0%)	9 (18.0%)	5 (10.0%)
MySQL	1 (2.1%)	12 (25.5%)	18 (38.3%)	2 (4.3%)
OpenLDAP	9 (18.4%)	4 (8.2%)	12 (24.5%)	12 (24.5%)

harden system against misconfigurations. The results may not sound impressive. However, if we consider the total number of configuration issues encountered in today's server systems, it is significant to eliminate around one third of the issues. Here, we consider *all* parameter-related configuration errors as the denominator. The percentages will be larger if we consider only one subtype such as illegal misconfigurations [185]. As a *first* step in the direction of improving configuration design, we believe that 24%–38% is a promising result.

To guide future research in this direction, Table 2.10 further breaks down the misconfiguration cases that cannot benefit from the tools. First, as discussed in §2.3.3, SPEX cannot infer all constraints. In addition, a configuration setting might conform to the constraints, but does not match the operators' intention. For example, a permission setting might be valid from the constraints' perspective, but insufficient for the operator to access files. Finally, even if the system already provides "good reactions" by our criteria (i.e., printing log messages containing the faulting parameters), operators might still report the problem because the semantics of the text messages might be confusing.

Table 2.11. Configuration constraints inferred by SPEX

Software	**Data type**		**Data range**	**Ctrl dep.**	**Value rel.**
	Basic	**Semantic**			
Storage-A	922	111	490	81	20
Apache	103	22	42	1	9
MySQL	272	74	213	35	10
PostgreSQL	231	52	186	44	6
OpenLDAP	75	15	20	0	2
VSFTP	130	34	84	68	1
Squid	258	46	120	14	9
Total	1991	354	1155	243	57

2.5.3 Configuration Constraint Inference

Table 2.11 breaks down different kinds of constraints inferred by SPEX. It infers a total of 3800 constraints from the evaluated systems. We can see that basic types can be inferred for most configuration parameters. In comparison, the number of semantic type is much smaller. SPEX cannot extract the semantic type for every parameter. It can only infer the semantic type if the parameter interacts with known APIs. Data range and inter-parameter correlations, especially control dependencies, are also common in the evaluated systems.

Table 2.12 shows the accuracy of constraint inference. We manually and carefully examined all of the 3800 constraints inferred by SPEX. SPEX achieves over 90% inference accuracy in most cases. We find that the inaccuracy is mainly caused by pointer aliasing. If a configuration parameter is pointed by aliased pointers, and/or there are complicated pointer arithmetic logic, SPEX may lose the correct mapping from the configuration parameter to the program variable, and thereby infer constraints that do not belong to the right parameter. Currently, SPEX does not perform any pointer-alias analysis. This explains why OpenLDAP has the lowest accuracy: many of its parameters are referenced through pointers. However, the overall accuracy is still over 90%, because most of the configuration parameters are not aliased.

Table 2.12. Accuracy of constraint inference

Software	**Data type**		**Data range**	**Ctrl dep.**	**Value rel.**
	Basic	**Semantic**			
Storage-A	97.0%	95.7%	87.1%	84.1%	94.1%
Apache	96.1%	91.7%	94.6%	100.0%	81.8%
MySQL	100.0%	98.7%	99.1%	94.7%	71.4%
PostgreSQL	100.0%	96.3%	97.3%	91.7%	85.7%
OpenLDAP	88.2%	93.7%	73.1%	N/A	50.0%
VSFTP	100.0%	100.0%	100.0%	63.9%	100.0%
Squid	77.0%	100.0%	100.0%	77.8%	100.0%

2.6 Experience and Practice

This section presents our experiences of reporting the misconfiguration vulnerabilities and error-prone configuration design and handling, as well as the good practices of configuration design and handling when we study the target software systems.

2.6.1 Interaction Experience with Developers

We reported the detected vulnerabilities and error-prone constraints to developers through the official bug reporting systems. To this day, 364 of the our reported vulnerabilities and 80 inconsistent constraints have been confirmed or fixed by the developers. The others are ignored or rejected by the developers, or being investigated. Here, we share our experience in interacting with developers.

Positive experience. We are encouraged by the positive feedback from many developers of the evaluated systems, and we appreciate the help from them.

- **Storage-A.** Misconfigurations account for one-third of the customer cases of Storage-A in this major U.S. storage company. It has imposed significant financial cost for troubleshooting these issues. Therefore, they actively investigate solutions to misconfigurations, and have been very supportive to our work, including providing us with source code, test cases, and allowing us to include Storage-A's results in this dissertation. All the exposed issues have been sent to the corresponding developing teams. Many of them have been fixed (c.f., Table 2.5), and others are under investigation.

- **Squid.** The developers immediately paid great attention to our reported misconfiguration vulnerabilities. We worked together and improved their configuration parsing library—adding more checks for configuration errors and more logging in reporting errors. Moreover, Squid developers applied our patches to make the case sensitivity consistent for 76 configuration parameters.

Negative experience. Not all interaction with developers is positive. Some of our reports and patches so far have been rejected or ignored by developers. The following summarize the typical negative responses. (1) Some developers think the information is clearly described in the document, so there is no need for systems to check or to pinpoint the configuration errors in log messages—"*The manual states, near the top...*" However, operators may not read manuals line by line, especially given that manuals for large systems are usually lengthy (e.g., MySQL-5.5's manual has 4502 pages). Also, operators may have problems understanding manual contents, because many operators come from a different background (c.f., Appendix A). (2) Some open-source developers tend to assume that operators read the source code (since it is open sourced) when they configure the system. In the response to one of our patches, the developer wrote, "*Most operators never adjust these values. Those who do read the code.*" Note that operators can read open-source code, but it does not mean that, operators have time, or are willing to read the code. (3) Some developers optimistically assume that operators would not make mistakes, "*If you work exactly and carefully it does not matter; if not, you should not maintain the server at all.*" As a result, it is not uncommon that developers closed the report with comments like "*This is not a bug.*" The implication is that "*the opera-*

tor must be a novice or not thinking." However, such optimistic assumptions are often proved unrealistic, as partially demonstrated in this dissertation and previous work on misconfiguration.

The negative experiences indicate that the battle to have developers take active roles in misconfiguration handling is challenging. The main impediment is the controversial responsibilities of the misconfiguration between operators and developers. Often times, it is only until the system suffers considerable support cost or failures (caused by misconfigurations) will the importance of active handling be appreciated by developers. We believe one way to raise this awareness is through education on user-friendly configuration design, hopefully leveraging the trend and attention in good user-interface design raised by Apple's success. As articulated in [113], developers should view system operators as their first-class users.

2.6.2 Practice

We highlight some of the good practices we observed from the evaluated software projects, and advocate adoption towards user-friendly configuration design.

Hiding critical configurations. Despite the trend that systems expose more and more configuration knobs, some systems choose to hide advanced and critical configuration parameters or options from operators, in order to avoid careless mistakes. Storage-A provides two levels of configuration interfaces: one for normal operators and the other for advanced ones. The former only exposes the common configurations while the latter exposes all the knobs. Moreover, it does not allow operators to directly modify system configuration files. The configuration settings from operators are enforced to go through the interfaces which perform basic checking. In fact, we observe that developers sometimes are struggling with the configurability. For example, eight Squid parameters have the following warning in their manual entries:

> *"Heavy voodoo here. I can't even believe you are reading this. Are you crazy? Don't even think about adjusting these unless you understand the algorithms in `comm_select.c` first!"*

A good practice should hide such esoteric parameters from operators, or forewarn operators with clear log messages when they are trying to configure these parameters. Chapter 4 discusses in detail on balancing the configurability (flexibility) and usability through user-centric configuration design.

Handling inconsistency. We observed two efforts in Storage-A in handling unit inconsistency. First, the unit information is included in the name of configuration parameters, e.g., "`cleanup.`*`msec`*" and "`takeover.`*`sec`*", which serves as both descriptions and mnemonics for operators. Second, some configuration parameters enforce operators to specify unit suffixes to help them express their intention explicitly.

Leveraging structural design. One idea is to provide systematic support to enforce configuration checking. Storage-A, MySQL, and PostgreSQL use global data structures to enforce developers to specify the data type and the min/max value for each configuration parameter. Based on these data structures, the systems easily enforce uniform validity checking of configuration settings. Consequently, they have fewer misconfiguration vulnerabilities that violate type and range constraints, as shown in Table 2.5.

2.7 Summary

In this chapter, we advocate the importance for software systems to anticipate and defend against misconfigurations. It makes a concrete useful step by providing tool support for developers to expose misconfiguration vulnerabilities, and detect error-prone configuration design and handling. SPEX has exposed 743 vulnerabilities and at least 112 error-prone constraints in both commercial and open-source systems. To this day, 364 vulnerabilities, together with 80 inconsistent constraints, have been confirmed or fixed by developers after we reported them. Our results have influenced the Squid Web proxy project to improve its configuration parsing library towards a more usable design. The original paper [179] has inspired research on improving system reactions to misconfigurations [92, 177, 193] and extracting configuration information through source code analysis [43, 132, 181, 196].

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the 24th ACM Symposium on Operating Systems Principles, 2013. Xu, Tianyin; Zhang, Jiaqi; Huang, Peng; Zheng, Jing; Sheng, Tianwei; Yuan, Ding; Zhou, Yuanyuan; Pasupathy, Shankar. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Early Detection

"*The early bird catches the worm.*"
—Proverb

Chapter 2 demonstrates that many misconfiguration vulnerabilities can be effectively exposed by testing the system's resilience through injecting configuration errors. However, vulnerabilities related to certain types of configurations, such as those of fault tolerance and error handling, are extremely hard to expose through testing. This is because these configurations are not needed at the initialization time or even during normal operations, but only used under critical circumstance (e.g., when the system encounters and has to deal with faults or failures). Therefore, errors in such configurations are often *latent* until their manifestations cause disasters.

One practical approach to tackling the dangerous latent configuration errors is to detect them as early as possible in order to minimize their failure damage rather than handling their consequences (similar to how we deal with fatal diseases like cancer). The following summarizes the importance of early detection of configuration errors:

- Early detection before configuration roll-out can prevent the same error from being replicated to thousands of nodes, especially in the data-center environment.

- Unlike software bugs, configuration errors, once detected, can be fixed by operators with no need to go through developers. Therefore, if detected earlier, the errors can be corrected immediately before the configurations are put online for production.

- For many configurations that control the system's failure handling, early detection of errors in their settings can prevent the system from entering an unrecoverable state (before any failures happen). Often, the combination of multiple errors (e.g., a configuration error together with a software bug) can bring down the entire service, as shown in many newsworthy outages [32, 152, 155, 158].

3.1 Introduction

This chapter explores the feasibility and efficacy of enabling early detection as an engineering practice through tool support to defense software systems against configuration errors, especially those latent ones.

It starts by understanding the root causes and characteristics of latent configuration (LC) errors. We study the practices of configuration checking in six mature, widely-deployed software systems, including HDFS, YARN, HBase, Apache, MySQL, and Squid. The study reveals that (1) many (14.0%–93.2%) of the critically important configuration parameters (those related to the system's reliability, availability, and serviceability) do not have any special code for checking the correctness of their settings. Instead, the correctness is verified (*implicitly*) when the configuration values are being actually used in operations; (2) many (12.0%–38.6%) of these configurations are not used at all during system initialization; (3) resulting from (1) and (2), 4.7%–38.6% of these critically important configuration parameters do not have any early checks and are thereby subject to LC errors that can severe impair the system's dependability.

To help systems detect LC errors early, we present a tool named PCHECK that analyzes the source code and automatically generates configuration checking code (termed *checkers*) for validating a system's configurations at its initialization phase. PCHECK takes a unique and intuitive method to check each configuration setting—*emulating the late execution that uses the configuration value; meanwhile capturing any anomalies exposed during the execution as the evidence of configuration errors*. PCHECK does not require developers to manually implement checking logic. The checkers generated by PCHECK are *generic*: they are not limited to any specific, predefined rule patterns, but are derived from how the program uses the parameters.

PCHECK shows that it is feasible to *accurately* and *safely* emulate late execution that uses configurations. It statically extracts the instructions that transform, propagate, and use the configuration values from the system program. To execute these instructions, PCHECK makes a best effort to produce the necessary execution context (values of dependent variables) that can be determined statically. PCHECK also "sandboxes" the emulated execution by instruction rewriting to prevent side effects on the running system or its environment.

More importantly, emulating the execution can expose many configuration errors as runtime anomalies (e.g., exceptions and error code) and the emulated execution runs in a short period. PCHECK inserts instructions to capture the anomalies that may occur during the emulated execution, as the evidence to report configuration errors.

As an enforcement, PCHECK encapsulates the emulated execution and error capturing code into checkers for every configuration parameter, and invokes the checkers at the system's initialization phase. This can minimize potential LC errors, and compensate for the missing and incomplete configuration checks in real-world systems.

3.2 Background

Unlike software bugs that typically go through various kinds of testing before releases (such as unit testing, regression testing, stress testing, system testing, etc.), operators often do not perform extensive testing on configurations before rolling them out to other nodes and putting the systems online [104]. Besides the lack of skills [104] and the temptation of convenience [97], the more fundamental reason is that operators do not have the same level of understanding on *how* and *when* the system uses each configuration value internally [103, 177]. Thus, they are limited to simple black-box testing such as starting the system and applying a few small workloads to see how the system behaves. Due to time and knowledge limitations, operators typically do not perform a comprehensive suite of test cases against configuration settings, especially for those hard-to-test ones (e.g., configurations related to fault tolerance and error-handling) that may require complex setups and even fault injections.

Therefore, early detection should inevitably fall onto the shoulder of the system itself—the system should automatically check as many configurations as possible at its early stages (the startup time). Unfortunately, many of today's systems either skip the checking or only check configurations right before the configuration values are used, as shown in our study (c.f., §3.3). Typically, at the startup time, only those configuration parameters needed for initialization are checked (or directly used), while many other parameters' checking is delayed much later until when they are used in special tasks. Since such configuration parameters are neither used nor checked during normal operations, errors in their settings go undetected until their late manifestation, e.g., under circumstances like error handling and fail-over. For simplicity, we refer to such errors as *latent configuration* (LC) errors.

3.2.1 Severity of Latent Configuration Errors

LC errors can result in severe failures, as they are often associated with configurations used to control critical situations such as fail-over [157], error handling [152], backup [142], load balancing [32], mirroring [158], etc. As explained above, their detec-

Table 3.1. Severity of latent vs. non-latent errors among the customers' configuration issues of COMP-A. LC errors contribute to 75% of high-severity configuration issues.

Severity level	**Latent**	**Non-latent**
All cases	47.6%	52.4%
High severity	75.0%	25.0%

Table 3.2. Diagnosis time of latent vs. non-latent errors among customers' configuration issues of COMP-A. The time is normalized by the average time of all the reported issues.

Error class	**Mean**	**Median**
Latent	1.14	1.70
Non-latent	0.87	0.41

tion or exposure is often too late to limit the failure damage. Take a real-world case as an example (c.f., §3.3: Figure 3.3a), an LC error in the fail-over configuration settings is detected only when the system encounters a failure (e.g., due to hardware faults or software bugs) and tries to fail-over to another component. In this case, the fail-over attempt also fails, making the entire system unavailable to all the clients.

Tables 3.1 and 3.2 compare the severity level and diagnosis time of real-world configuration issues caused by LC errors versus non-latent configuration errors (detected at system startup) of COMP-A[1], a major storage company in the US. Although there have been fewer LC errors than non-latent ones, LC errors contribute to 75% of the high-severity issues and take longer to diagnose, indicating their high impact and damage.

Figure 3.1 shows a real-world LC error from Squid, a widely used open-source Web proxy server. The LC error resided in `diskd_program`, a configuration parameter used only during log rotation. Squid did not check the configuration during initialization; thus, this error was exposed much later after days of execution. It caused 7+ hours of system downtime and cost 48 hours of diagnosis efforts. After the error was finally discerned, the Squid developers added a patch to proactively check the setting at system startup time to prevent such latent failures.

[1]We are required to keep the company and its products anonymous.

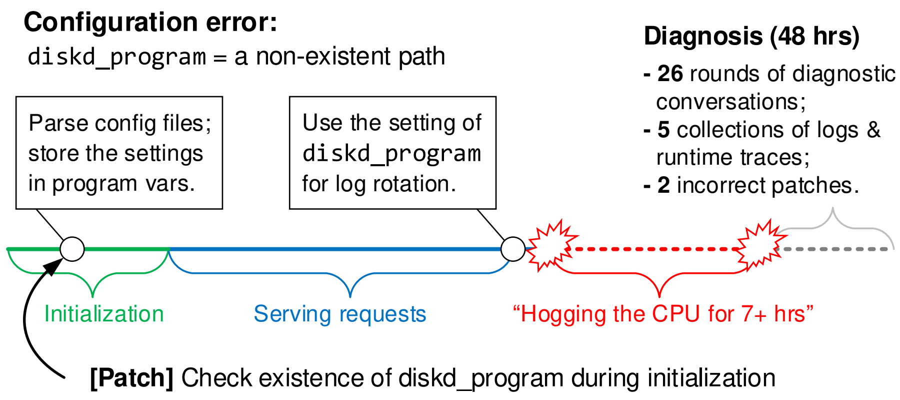
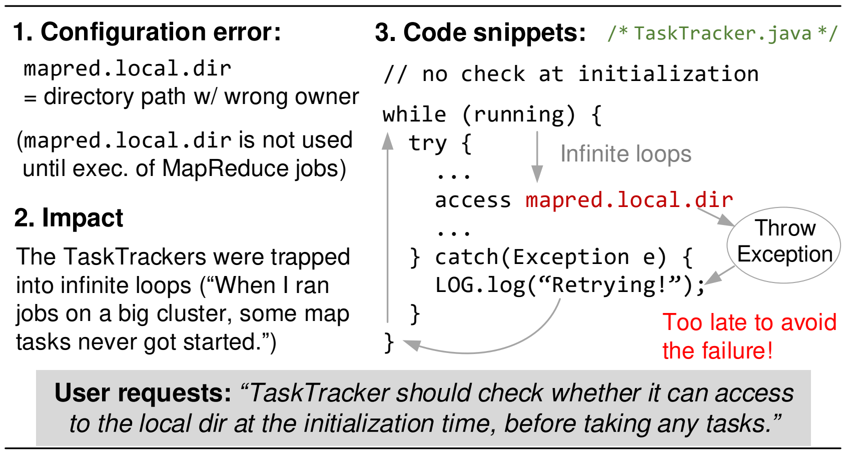

Configuration error:
diskd_program = a non-existent path
Parse config files; store the settings in program vars.
Use the setting of diskd_program for log rotation.
Diagnosis (48 hrs)
- 26 rounds of diagnostic conversations;
- 5 collections of logs & runtime traces;
- 2 incorrect patches.
Initialization
Serving requests
"Hogging the CPU for 7+ hrs"
[Patch] Check existence of diskd_program during initialization

Figure 3.1. A real-world LC error from Squid [142]. The error caused system hanging for 7+ hours, and resulted in 48 hours of diagnosis efforts. Later, a patch was added to check the existence of the configured path during initialization. Unfortunately, the patched check is still subject to LC errors such as incorrect file types and permissions.


1. Configuration error:
mapred.local.dir
= directory path w/ wrong owner
(mapred.local.dir is not used until exec. of MapReduce jobs)
2. Impact
The TaskTrackers were trapped into infinite loops ("When I ran jobs on a big cluster, some map tasks never got started.")
3. Code snippets:
Infinite loops
Throw Exception
Too late to avoid the failure!
User requests: "TaskTracker should check whether it can access to the local dir at the initialization time, before taking any tasks."

```
/* TaskTracker.java */
// no check at initialization
while (running) {
  try {
    ...
    access mapred.local.dir
    ...
  } catch(Exception e) {
    LOG.log("Retrying!");
  }
}
```

Figure 3.2. A real-world LC error from MapReduce [59]. When the exception handler caught the runtime exception induced by the LC error, it was already too late to avoid the downtime. After this incident, the reporter requested to check the configuration "at the initialization time."

Figure 3.2 shows another real-world example in which an LC error failed a large-scale MapReduce job processing. This LC error was replicated to multiple nodes and crashed the TaskTrackers on those nodes. Specifically, the configuration error caused a runtime exception on each node. The TaskTracker caught the exception and restarted the job. Unfortunately, as the error is persistent in the configuration file, restarting the job failed to get rid of the error but induced infinite loops. Note that when the exception handler caught the error, it was already too late to avoid downtime (the best choice is to terminate the jobs).

Preventing above LC-error issues would require software systems to check configurations *early* during the initialization time, even though the configuration values are only needed in much later execution or during special circumstances. This is indeed demonstrated by the developers' postmortem patches. As revealed in Facebook's recent study [155], 42% of the configuration errors that caused high-impact incidents are "obvious" errors (e.g., typos), indicating the limitations of code review and system testing in preventing LC errors. These errors might be detected by early checks (only if developers are willing to and remember to write the checking code).

3.2.2 Limitation of Existing Detection Approaches

Most of the existing detection tools check configuration settings against apriori correctness rules (known as *constraints*). However, as large software systems usually have hundreds to thousands of configuration parameters, it is time-consuming and error-prone to ask developers to *manually* specify every single constraint, not to mention that constraints change with software evolution [192].

So far, only a few automatic configuration-error detection tools have been proposed. Most of them detect errors by *learning* the "normal" values from large collections of configuration settings in the field [115, 131, 189, 190]. While these techniques are effective in certain scenarios, they have the following limitations, especially when being applied to cloud and data centers.

First, most of these works require a large collection of *independent* configuration settings from hundreds of machines. This is a rather strong requirement, as most cloud and data centers typically propagate the same configurations from one node to all the other nodes. Thereby, the settings from these nodes are not independent, and thus not useful for "learning". Second, they do not work well with configurations that are inherently different from one system to another (e.g., domain names, file paths, IP addresses) or incorrect settings that fall in normal ranges. They also cannot differentiate customized settings from erroneous ones. Furthermore, most of these tools target on specific error types (encoded by their predefined constraint templates) and are hard to generalize to detect other types of errors. A recent work learns constraints from KB (Knowledge Base) articles [118]. However, this approach has the same limitations discussed above. Specially, KB articles are mainly served for postmortem diagnosis and thus may not cover every single constraint.

There are very few configuration-error detection approaches that do not rely on constraints specified manually by developers or learned from large collections of independent settings (or KB articles). The only exception (to the best of our knowledge) is CONF_SPELLCHECKER which detects simple errors based on type inference from source code. While this technique is very practical, it is limited in the types of configuration errors that can be detected, as shown in our experimental evaluation (§3.5).

3.3 Understanding Latent Configuration Errors

To understand the root causes and characteristics of LC errors, we study the practices of the configuration checking and error detection in six mature, widely-deployed open-source software systems (c.f., Table 3.3). They cover multiple functionalities and languages, and include both single-machine and distributed systems.

We focus on configuration parameters used in components related to the system's Reliability, Availability, and Serviceability (known as RAS for short [140]). For each system considered, we select all the configuration parameters of RAS-related features based on the software's official documents, including error handling, fail-over, data

Table 3.3. The systems and the RAS parameters studied in §3.3.

Software	Description	Lang.	# Parameters	
			Total	**RAS**
HDFS	Dist. filesystem	Java	164	44
YARN	Data processing	Java	116	35
HBase	Distributed DB	Java	125	25
Apache	Web server	C	97	14
Squid	Proxy server	C/C++	216	21
MySQL	DB server	C++	462	43

backup, recovery, error logging and notification, etc. The last column of Table 3.3 shows the number of the studied RAS parameters. Compared with configurations of other system components, configurations used by RAS components are more likely to be subject to LC errors due to their inherently latent nature; moreover, the impact of errors in RAS configurations is usually more severe.

Note: LC errors are not limited to RAS components. Thus, the reported numbers may not represent the overall statistics of all the LC errors in the studied systems. In addition, PCHECK, the tool presented in §3.4, applies to all the configuration parameters; it does not require manual efforts to select out RAS parameters.

3.3.1 Methodology

We manually inspect the source code related to RAS configuration parameters of the studied systems. First, for each RAS parameter, we study the code that checks the parameter setting at the system's initialization phase[2] (if any) and the code that later uses the parameter's value. Then, we compare these two sets of code (checking vs. usage) and examine if the initial checking is sufficient to detect configuration errors. If an error can escape from the initialization phase and break the usage, it is a potential LC error.

We verify each LC error discovered from source code by exposing and observing

[2]A system's initialization phase is defined from its entry point to the point it starts to serve user requests or workloads.

Table 3.4. The number of configuration parameters that do not have any initial checking code ("missing") and that only have partial checking and thus cannot detect all potential errors ("incomplete").

Software	**Deficiency of initial checking**		**# Studied parameters**
	Missing	**Incomplete**	
HDFS	41 (93.2%)	3 (6.9%)	44
YARN	29 (82.9%)	5 (14.3%)	35
HBase	18 (72.0%)	5 (2.0%)	25
Apache	4 (28.6%)	2 (14.3%)	14
Squid	9 (42.9%)	4 (19.0%)	21
MySQL	6 (14.0%)	6 (14.0%)	43

the impact of the error. We first inject the errors into the system's configuration files and launch the system; then we trigger the manifestation conditions to expose the error impact. For example, to verify the LC errors in the HDFS auto-failover feature, we start HDFS with the erroneous fail-over settings, trigger the fail-over procedure by killing the active NameNode, and examine if the fail-over can succeed. As all the LC errors are verified through their manifestation, there is no false positive in the reported numbers.

3.3.2 Findings

Finding 1: *Many (14.0%–93.2%) of the studied RAS parameters do not have any special code for checking the correctness of their settings. Instead, the correctness is verified (implicitly) when the parameters' values are actually used in operations.*

Table 3.4 shows the number of the studied RAS parameters that rely on the usage code for verifying correctness, because their initial checks are either *missing* or *incomplete*. Most of the studied RAS parameters in HDFS, YARN, and HBase do not have any special code for checking the correctness of their settings. These systems adopt the *lazy* practice of using configuration values[3]—parsing and consuming configuration

[3]This is a bad but commonly adopted practice in Java and Python programs which rely on libraries (e.g., `java.util.Properties` and `configparser`) to directly retrieve and use configuration values from configuration files on demand, without systematic early checks.

settings only when the values are immediately needed for the operations, without any systematic configuration checking at the system's initialization phase.

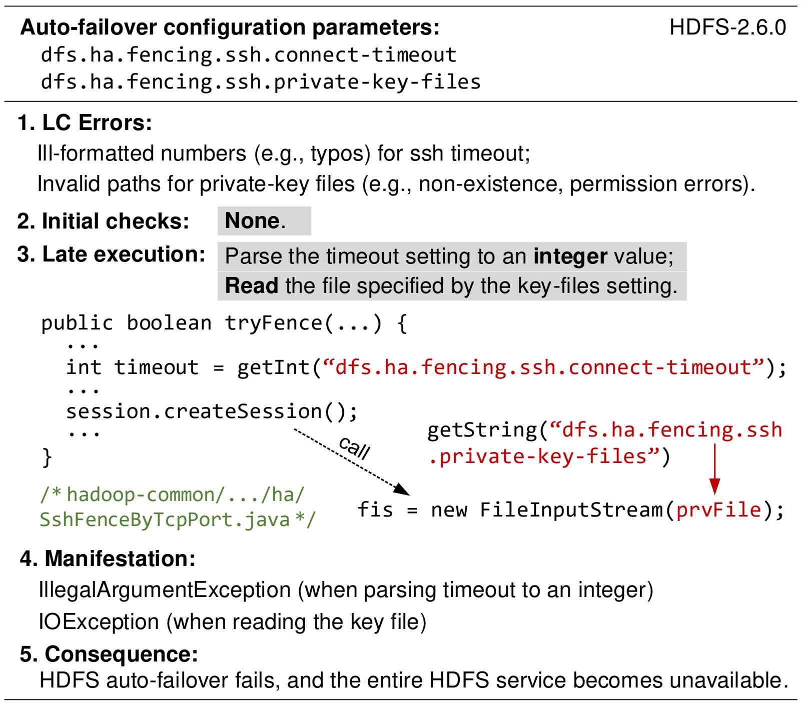
With such a practice, even a trivial error could result in disastrous impact on the system's dependability. Figure 3.3a exemplifies such cases using the new LC errors we discovered in our study. In HDFS, any LC errors (such as a naïve type error) in the auto-failover configurations could break the fail-over procedure upon the NameNode failures (as the values are not checked or used early). As a consequence, the entire HDFS service would become unavailable.

Apache, MySQL, and Squid all apply specific configuration checking procedures at initialization, mainly for checking data types and data ranges. However, for more complicated parameters, some checking is incomplete. Figure 3.3b shows another new LC error we discovered. In this case, though the initial checking code covers file existence and types, it misses other constraints such as file permissions. This leaves Apache subject to permission-related LC errors (which is reported as one common cause of core-dump failures upon server crash [148]).

As shown by Figure 3.3b, one configuration parameter could have multiple subtle constraints depending on how the system uses its value. For example, a configured file path used by `chdir` has different constraints from files accessed by `open`; even for files accessed by the same `open` call, different flags (e.g., `O_RDONLY` versus `O_CREAT`) would result in different constraints. Implementing code manually to check such constraints is tedious and error-prone.

Finding 2: *Many (12.0%–38.6%) of the studied RAS configuration parameters are not used at all during the system's initialization phase.*

Table 3.5 counts the studied configuration parameters that are not used at the system's initialization phase, but are consumed directly in late execution (e.g., when



Auto-failover configuration parameters: HDFS-2.6.0
`dfs.ha.fencing.ssh.connect-timeout`
`dfs.ha.fencing.ssh.private-key-files`

1. LC Errors:
Ill-formatted numbers (e.g., typos) for ssh timeout;
Invalid paths for private-key files (e.g., non-existence, permission errors).

2. Initial checks: **None**.

3. Late execution: Parse the timeout setting to an **integer** value;
Read the file specified by the key-files setting.

```
public boolean tryFence(...) {
  ...
  int timeout = getInt("dfs.ha.fencing.ssh.connect-timeout");
  ...
  session.createSession();
  ...
}
/* hadoop-common/.../ha/
SshFenceByTcpPort.java */
```

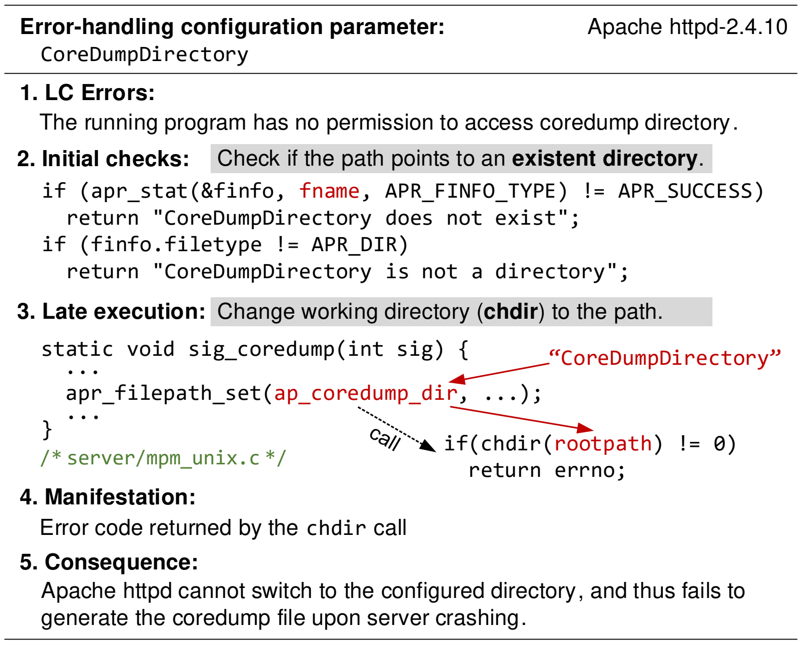
```
getString("dfs.ha.fencing.ssh
.private-key-files")

fis = new FileInputStream(prvFile);
```

4. Manifestation:
IllegalArgumentException (when parsing timeout to an integer)
IOException (when reading the key file)

5. Consequence:
HDFS auto-failover fails, and the entire HDFS service becomes unavailable.

(a) Missing initial checking



Error-handling configuration parameter: Apache httpd-2.4.10
`CoreDumpDirectory`

1. LC Errors:
The running program has no permission to access coredump directory.

2. Initial checks: Check if the path points to an **existent directory**.

```
if (apr_stat(&finfo, fname, APR_FINFO_TYPE) != APR_SUCCESS)
  return "CoreDumpDirectory does not exist";
if (finfo.filetype != APR_DIR)
  return "CoreDumpDirectory is not a directory";
```

3. Late execution: Change working directory (**chdir**) to the path.

```
static void sig_coredump(int sig) {
  ...
  apr_filepath_set(ap_coredump_dir, ...);
  ...
}
/* server/mpm_unix.c */
```

```
"CoreDumpDirectory"

if(chdir(rootpath) != 0)
  return errno;
```

4. Manifestation:
Error code returned by the `chdir` call

5. Consequence:
Apache httpd cannot switch to the configured directory, and thus fails to generate the coredump file upon server crashing.

(b) Incomplete initial checking

Figure 3.3. New LC errors discovered in the latest versions of the studied software, both of which are found to have caused real-world failures [147, 148]. For all these LC errors, the correctness checking is implicitly done when the parameters' values are actually used in operations, which is unfortunately too late to prevent the failures.

Table 3.5. The studied configuration parameters whose values are not used at the system's initialization phase.

Software	Not used during initialization	# Studied parameters
HDFS	17 (38.6%)	44
YARN	9 (25.7%)	35
HBase	3 (12.0%)	25
Apache	4 (28.6%)	14
Squid	4 (19.0%)	21
MySQL	6 (13.9%)	43

dealing with failures). Figure 3.3a is such an example. Since all these parameters are from RAS features, it is natural for their usage to come late on demand.

Some Java programs put the checking or usage code of the parameters in the class constructors, so that the errors can be exposed when the class objects are created (specially, this is used as the practice for quickly fixing LC errors [65,66,182]). However, this may not fundamentally avoid LC errors if the class objects are not created during the system's initialization phase.

Note: RAS configurations can be implemented with early usage at the system's initialization phase. As shown in Table 3.5, the majority of RAS configurations are indeed used during initializaiton. For example, all the studied systems choose to `open` error-log files at initialization time, rather than waiting until they have to print the error messages to the log files upon failures.

Finding 3: *Resulting from Findings 1 and 2, 4.7%–38.6% of the studied RAS parameters do not have any early checks and are thereby subject to LC errors which can cause severe impact on the system's dependability.*

Table 3.6 shows the number of the RAS configuration parameters that are subject to LC errors in each studied system. The threats are prevalent: LC errors can reside in 10+% of the RAS parameters in five out of six systems. As all these LC errors are discovered in the latest versions, any of them could appear in real deployment and

Table 3.6. The number of configuration parameters that are subject to LC errors in the studied ones. 11 of these parameters have been confirmed/fixed by the developers after we reported them.

Software	**# RAS Parameters**	
	Subject to LC errors	**# Studied parameters**
HDFS	17 (38.6%)	44
YARN	9 (25.7%)	35
HBase	3 (12.0%)	25
Apache	3 (21.4%)	14
Squid	3 (14.3%)	21
MySQL	2 (4.7%)	43
Total	**37 (20.3%)**	**182**

would impair the system's dependability in a latent fashion. Such prevalence of LC errors indicates the need for tool support to systematically rule out the threats.

Among the studied systems, HDFS and YARN have a particularly high percentage of RAS parameters subject to LC errors, due to their lazy evaluation of configuration values (refer to Finding 1 for details). HBase applies the same lazy practice as HDFS and YARN, but has fewer parameters subject to LC errors, because most of its RAS parameters are used during its initialization. We also find LC errors in the other studied systems, despite their initial configuration checking efforts.

3.3.3 Implication

In summary, even mature software systems are subject to LC errors due to the deficiency of configuration checking at the initialization time. While relying on developers' discipline to add more checking code can help, the reality often fails our expectations, because implementing configuration checking code is tedious and error-prone.

Fortunately, we also observe from the study that except for *explicit* configuration checking code, the actual *usage* of configuration values (which already exists in source code) can serve as an *implicit* form of checking, for example, `open`ing a file path that comes from a configuration value implies a capability check. Such usage-implied checking is often more complete and accurate than the explicit checkers written by developers, because it precisely captures how the configuration values should be used in the actual program execution. Sadly, in reality these usage-implied checking is rarely leveraged to detect LC errors, because the usage often comes too late to be useful. A natural question regarding the solution to LC errors is: can we automatically generate configuration checking code from the existing source code that uses configuration values?

3.4 PCHECK Design and Implementation

PCHECK is a tool for enabling early detection of configuration errors for a given systems program. The objective of PCHECK is to automatically generate configuration checking code (termed *checkers*) based on the original program, and invoke them at the system initialization phase, in order to detect LC errors.

PCHECK tries to generate checkers for every configuration parameter. It is *not* specific to RAS configurations and has *no* assumption on the existence of any LC errors. The checker of a parameter emulates how the system uses the parameter's value in the original execution, and captures anomalies exposed during the emulated execution as the evidence of configuration errors.

PCHECK is built on top of the Soot [3] and LLVM [5] compiler frameworks and works for both Java and C system programs. PCHECK works on the intermediate representations (IR) of the programs (LLVM IR or Soot Jimple). It takes the original IR as inputs, and outputs the generated checkers, and inserts them into bitcode/bytecode files (which are then built into native binaries). This may require prepending the build process by replacing the compiler front-end with Soot or Clang [167].

PCHECK faces three major challenges: (1) How to automatically emulate the execution that uses configuration values? (2) Since the checkers will be inserted into the original program and will run in the same address space, how does one make the emulation *safe* without incurring side effects on the system's internal state and external environment? (3) How to capture anomalies during the emulated execution as the evidence of configuration errors (the emulation alone cannot directly report errors)?

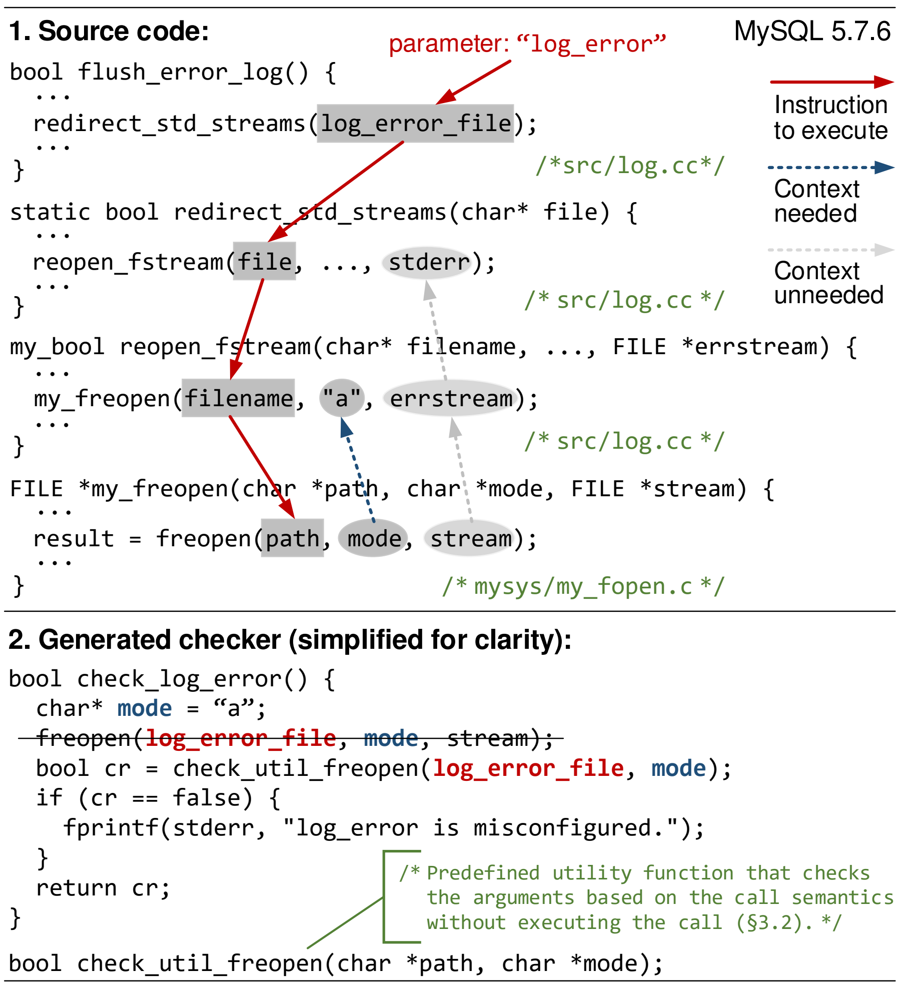
To address the first challenge, PCHECK extracts the instructions that transform, propagate, and use the value of every configuration parameter using a static taint tracking method. PCHECK then makes a best effort to produce the context (values of dependent variables) necessary for emulating the execution. The extracted instructions, together with the context, are encapsulated in a checker.

For the second challenge, PCHECK "sandboxes" the auto-generated checkers by rewriting instructions that would cause side effects. PCHECK avoids modifications to global variables by copying their values to local ones, and rewrites the instructions that may have external side effects on the underlying OS.

To address the third challenge, PCHECK leverages system- and language-level error identifiers (including runtime exceptions, system-call error codes, and abnormal program exits) to capture the anomalies exposed during the emulation, as the evidence to report configuration errors.

Figure 3.4 illustrates PCHECK's checker generation for a MySQL configuration parameter, `log_error`, which is subject to LC errors [105]. PCHECK extracts the instructions that use the configuration value and determines the values of the other dependent variables (e.g., `mode`) as the context. To prevent side effects, it rewrites some call instruction. It detects errors based on the return value. Lastly, PCHECK inserts the generated checkers into the system program, and invokes these checkers at the end of the system initialization phase (annotated by developers). To detect TOCTTOU errors[4], PCHECK supports running checkers periodically in a separate thread.

[4]A TOCTTOU (Time-Of-Check-To-Time-Of-Use) error occurs after the checking phase and before the use phase, e.g., inadvertently deleting a file that had been checked early but will be used later on.



```
bool flush_error_log() {
  ...
  redirect_std_streams(log_error_file);
  ...
}                                          /*src/log.cc*/
static bool redirect_std_streams(char* file) {
  ...
  reopen_fstream(file, ..., stderr);
  ...
}                                         /* src/log.cc */
my_bool reopen_fstream(char* filename, ..., FILE *errstream) {
  ...
  my_freopen(filename, "a", errstream);
  ...
}                                         /* src/log.cc */
FILE *my_freopen(char *path, char *mode, FILE *stream) {
  ...
  result = freopen(path, mode, stream);
  ...
}                                     /* mysys/my_fopen.c */
```

```
bool check_log_error() {
  char* mode = “a”;
  ~~freopen(log_error_file, mode, stream);~~
  bool cr = check_util_freopen(log_error_file, mode);
  if (cr == false) {
    fprintf(stderr, "log_error is misconfigured.");
  }
  return cr;               /* Predefined utility function that checks
}                             the arguments based on the call semantics
                              without executing the call (§3.2). */
bool check_util_freopen(char *path, char *mode);
```

Figure 3.4. Illustration of PCHECK’s checker generation (using a real-world LC error example [105]). PCHECK replaces the original call (`freopen`) with check utilities based on `access` and `stat` to prevent side effects (§3.4.2). To execute the instructions, the necessary execution context needs to be produced. Note that we illustrate the checker using C code for clarity; the actual code is in LLVM IR or Soot Jimple.

Usage. PCHECK requires two inputs from developers: (1) *specifications of the configuration interface* to help PCHECK identify the initial program variables that store configuration values, as the starting points for analysis (§3.4.1); (2) *annotations* of the system’s initialization phase where the early checkers will be invoked (§3.4.4).

In addition, PCHECK provides the tuning interface for developers to select and remove any generated checkers, as per their preference and criteria (e.g., after standard software testing of the enhanced system programs). Similarly, PCHECK provides an operational interface that allows operators to enable/disable the invocation of the checkers of any specific parameters in operation.

3.4.1 Emulating Execution

To emulate the execution that uses a configuration parameter, PCHECK first identifies instructions that load the parameter's value into program variables (§3.4.1). Starting from there, PCHECK performs forward static taint analysis to extract all the instructions whose execution uses the parameter's value, and hence are the candidates to be included in the checkers (§3.4.1). It then analyzes backwards to figure out the values of dependent variables in these instructions, as the execution context (§3.4.1). Finally, PCHECK composes checkers using the above instructions and their context (§3.4.1).

Note that the emulation does not need to include the conditions under which the configurations are used. Instead, it focuses on executing the instructions that consume the configuration values—the goal is to check if using the configuration would cause any anomalies when its value is needed. With this design, PCHECK is able to effectively handle large, non-deterministic software programs, without the need to inject/simulate hard-to-trigger error conditions under which LC errors are exposed.

Identifying Starting Points

As the configuration-consuming execution always starts from loading the configuration value, PCHECK needs to identify the program variables that initially store the value of each parameter, as the starting points. PCHECK adopts the common practices described in §2.3.2 (Chapter 2) to obtain the mapping from configuration parameters to the corresponding variables. As a recap, the basic idea is to let developers specify the *interface*[5] for retrieving configuration values, and then automatically identify program variables that load the values based on the interface. As discussed in §2.3.2, most mature systems have uniform configuration interfaces for the ease of code maintenance. For instance, to work with HDFS, PCHECK only needs to know the configuration getter functions (e.g., `getInt` and `getString` in Figure 3.3a) declared in a single Java class; identifying them only requires several lines of specifications using regular expressions. In general, specifying interface requires little specification efforts, compared to annotating every single variable for a large number of configuration parameters. In the evaluation, the specifications needed for most systems are less than 10 lines (§3.5).

Extracting Instructions Using Configurations

For each configuration parameter, PCHECK extracts the instructions that propagate, transform, and use the parameter's value using a static taint tracking method. For a given parameter, the initial taints are the program variables that store the parameter's value (§3.4.1). The taints are propagated via data-flow dependencies (including assignments, type casts, and arithmetic/string operations), but not through control-flow dependencies to avoid overtainting [144]. All the instructions containing taints are extracted, and will be encapsulated in a checker.

Note that one parameter could be used in multiple execution paths, and thus have multiple checkers. We explain how multiple checkers are aggregated in §3.4.1.

Ordinarily, the extracted instructions from data-flow analysis do not include branches. However, if a tainted instruction is used as a branch condition whose branch body encloses other tainted instructions, PCHECK performs additional control-flow analysis to retain the control dependency of these instructions. One pattern is using a configuration value `p` after a null-pointer check, in the form of, `if (p != NULL) { use`

[5]The interface could be APIs, data structures, or parsing functions (c.f., §2.3.2). As discussed in §2.3.2, only three types of interfaces are commonly used to store/retrieve configurations.

`p; }`. PCHECK recovers the conditional branch and ensures that if `p`'s value is `NULL`, the instructions using `p` inside the branch would not be reached. Moreover, PCHECK checks if a tainted branch condition leads to abnormal program states, for which it inserts error-reporting instructions (see §3.4.3).

The taint tracking is inter-procedural, context sensitive, and field sensitive. Inter-procedure is necessary because configuration values are commonly passed through procedure calls, as illustrated in Figure 3.4. We adopt a summary-based inter-procedural analysis, and assemble the execution based on arguments/returns. PCHECK maintains the call sites; thus it naturally enables context sensitivity which helps produce context by backtracking from callees to callers (c.f., §3.4.1). Field sensitivity is needed as configuration values could be stored in data structures or as class fields. PCHECK scales well for real-world software systems, as configuration-related instructions form a small part of the entire code base. We do not explicitly perform alias analysis (though it is easy to integrate), as configuration variables are seldom aliased.

Producing Execution Context

Some of the extracted instructions that use configuration variables may not be directly executable, if they contain variables that are not defined within the extracted instruction set. To execute such instructions, PCHECK needs to determine the values of these undefined variables (which we refer to as "dependent variables") in order to produce self-contained context.

PCHECK will include a variable and the corresponding instructions in the emulated execution, only when this variable's value stems from configuration values (e.g., `path` in Figure 3.4) or can be statically determined along the data-flow paths of the configuration value (e.g., `mode` and `stream` in Figure 3.4). PCHECK does not include dependent variables whose values come from indeterminate global variables, external inputs (from I/O or network operations such as `read` and `recv`), values defined out of the scope of the starting point, etc. For such dependent variables, PCHECK removes the instruction that uses them as operands, together with the succeeding instructions. Those variables' values may not be available during the initialization phase of the system execution; using them would lead to unexpected results.

To produce the context, PCHECK backtracks each undefined dependent variable first intra-procedurally and then inter-procedurally (to handle the arguments of procedure calls). The backtracking starts from the instruction that uses the variable as its operand, and stops until either PCHECK successfully determines the value of the variable or gives up (the value is indeterminate). In Figure 3.4, PCHECK backtracks `mode` used by the tainted instruction and successfully obtains its value `"a"`.

PCHECK only attempts to produce the minimal context necessary to emulate execution for the purpose of checking. As an optimization, PCHECK is aware of how certain types of instructions will be rewritten in later transformations (e.g., for side-effect prevention, §3.4.2). In Figure 3.4's example, PCHECK knows how the `freopen` call will be rewritten later. Therefore, it only produces the context of `mode` which is needed to check the file access; the other dependent variable `stream` is ignored as it is not needed for the checking.

Sometimes, the dependent variables come from other configuration parameters. PCHECK can capture the relationships among multiple configurations, e.g., one parameter's value has to be larger or smaller than another's.

Encapsulation

For each configuration parameter, PCHECK encapsulates the instructions that consume configuration values together with their context into a *checker*, in the form of a *function*. PCHECK clones the original instructions and their operands. For local variables used as operands, PCHECK clones a new local variable and replaces the original variable with the new one. If the instructions change global variables, PCHECK generates a corresponding local variable and copies the global variable's value to the local one (to avoid changing the global program state). When it involves procedure calls, PCHECK inlines the callees.

Handling multiple execution paths. For configuration parameters whose values are used in multiple distinct execution paths, PCHECK generates multiple checkers and aggregates their results. The configuration value is considered erroneous if one of these checkers complains. PCHECK needs to pay attention to potential path explosion to avoid generating too many checkers. Fortunately, in our experience, configuration values are usually used in a simple and straightforward way, with only a small number of different execution paths to emulate.[6] This makes the PCHECK approach feasible.

Moreover, PCHECK merges two checkers if they are equivalent or if one is equivalent to a subset of the other. PCHECK does this by canonicalizing and comparing the instructions in the checkers' function bodies. Additionally, PCHECK merges checkers which start with the same transformation instruction sequence by reusing the intermediate transformation results.

Note that the checkers with no error identifiers (§3.4.3) or considered redundant (§3.4.4) will be abandoned. As shown in §3.5.4, the number of generated checkers are well bounded, and executing them incurs little overhead.

3.4.2 Preventing Side Effects

PCHECK ensures that the generated checkers are free of side effects—running the checkers does not change the *internal* program state beyond the checker function

[6]The emulated execution paths are not the original execution paths (they only include the configuration-related instructions).

itself, or the *external* system environment (e.g., filesystems and OSes). Therefore, PCHECK cannot blindly execute the original instructions. For example, if the checker contains instructions that call `exec`, running the checker would destruct the current process image. Similarly, creating or deleting files is not acceptable, as the filesystem state before and after checking would be inconsistent.

Internal side effects are prevented by design. PCHECK ensures that each checker only has local effects. As discussed in §3.4.1, PCHECK avoids modifying global variables in the checker function; instead, it copies global variable values to local variables and uses the local ones instead. The checker does not manipulate pointers if the pointed values are indeterminate.

External side effects are mainly derived from certain system and library calls that interact with the external environment (e.g., filesystems and OS states). In order to preserve the checking effectiveness without incurring external side effects, PCHECK rewrites the original call instructions to redirect the calls to predefined *check utilities*. A check utility *models* a specific system or library call based on the call semantics. It validates the arguments of the call, but does not actually execute the call. PCHECK implements check utilities for standard APIs and data structures (including system calls, `libc` functions for C, and Java core packages defined in SDK). The check utilities are implemented as libraries that are either statically linked into the system's bitcode (for C programs), or included in the system's `classpath` (for Java programs). In Figure 3.4, the check utility of `freopen` checks the arguments of the call using `access` and `stat` which are free of side effects (the original `freopen` call will close the file stream specified by the third argument).

PCHECK skips instructions that `read`/`write` file content or `send`/`recv` network packets, in order to stay away from external side effects and heavy checking overhead. Instead, PCHECK performs metadata checks for files and reachability checks for network addresses. This helps the generated checkers be safe and efficient, while still being able to catch a majority of real-world LC errors. For any library calls that are not defined in PCHECK or do not have known side effects (e.g., some library calls would invoke external programs/commands), PCHECK defensively removes the call instructions (together with the succeeding instructions) to avoid unexpected effects.

One alternative to preventing external side effect is to running the checkers inside a sandbox or even a virtual machine at the system initialization phase. This may save the efforts of implementing the check utilies and rewriting system/library call instructions. However, such approach would impair the usability of PCHECK, because it requires additional setups from operators in order to run the PCHECK-enhanced program.

3.4.3 Capturing Anomalies

As the checker emulates the execution that uses the configuration value, anomalies exposed during execution indicate that the value contains errors—the same problem that would occur during real execution. In this case, the checker reports errors and pinpoints the parameter.

PCHECK captures anomalies based on the following three types of *error identifiers*: (1) runtime exceptions that disrupt the emulated execution (for Java programs); (2) error code returned by system and library calls (for C programs); and (3) abnormal program termination and error logging that indicate abnormal program states.

For Java programs, PCHECK captures the runtime anomalies based on Java's `Exception` interface, the language's uniform mechanism for capturing error events. PCHECK places the body of the checker function in a `try`/`catch` block. The abnormal execution would throw `Exception` objects and fall into the `catch` block. In this case, the checker reports errors and prints the stack traces.

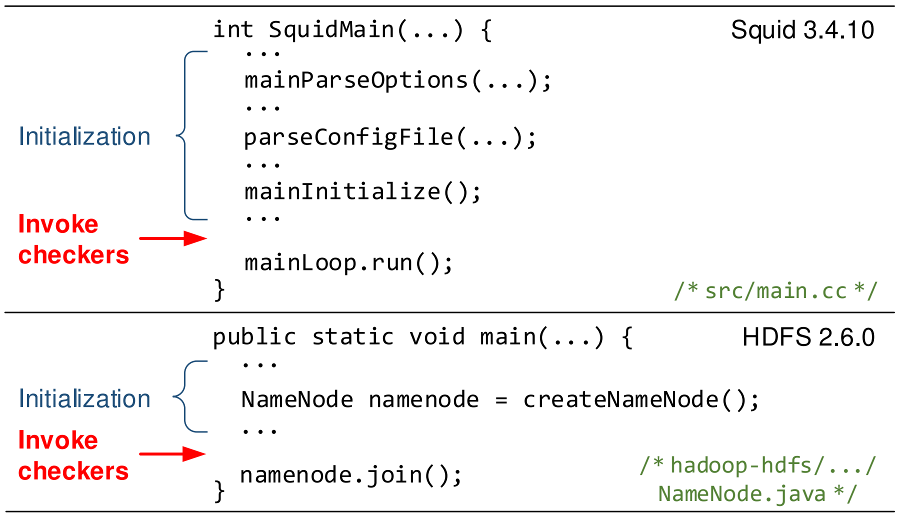
C programs do not have the uniform error interfaces. Thus, PCHECK leverages the error identifiers defined by specific system-/library-call semantics, i.e., the return values and `errno`. For example, if the `access` call returns `-1`, it means the call failed when accessing the file (with the reason being encoded in `errno`). In PCHECK, we predefine the error identifiers for commonly-used system and `libc` calls to decide whether a call succeeded or failed. If the call fails, the checker reports errors.

In addition to the anomalies exposed by system and library APIs, a program usually contains hints of abnormal program states. Such hints are instructions such as `exit`, `abort`, `throw`, false assertion, error logging, etc. PCHECK treats these hints as one type of anomalies. If an instruction is post-dominated by any anomaly hints, the instruction itself indicates an abnormal state of execution. Thus, PCHECK reports configuration errors when the checker emulates such error instructions. PCHECK records these hints during the code analysis in §3.4.1, and inserts error-reporting instructions into the checker at the corresponding locations.

PCHECK abandons the checkers that do not contain any of the three types of error identifiers discussed above. In other words, running such checkers cannot expose any explicit anomalies (no evidence of configuration errors).

3.4.4 Invoking Early Checkers

Once the checkers are generated, PCHECK inserts call instructions to invoke the checkers at the program locations specified by developers. The expected location is at the end of the system initialization phase to make the checkers the last defense against LC errors. Figure 3.5 shows the locations annotated for PCHECK to invoke the auto-generated checkers for Squid and HDFS. For server systems like Squid, the checkers should be invoked before the server starts to listen and wait for client requests. For distributed systems like HDFS, the checkers should be invoked before the system starts to connect and join the cluster. As all the evaluated systems fall in these two patterns,



```
int SquidMain(...) {
   ...
   mainParseOptions(...);
   ...
   parseConfigFile(...);
   ...
   mainInitialize();
   ...

   mainLoop.run();
}                                    /* src/main.cc */
```

```
public static void main(...) {
   ...
   NameNode namenode = createNameNode();
   ...
                                     /* hadoop-hdfs/.../
   namenode.join();
}                                      NameNode.java */
```

Figure 3.5. Locations to invoke the checkers in Squid and HDFS NameNode. The auto-generated checkers are expected to be invoked at the end of the initialization phase.

we believe that specifying the invocation locations is a simple practice for developers.

Some C programs may change user/group identities. Typically, the program starts as `root` and then switches to unprivileged users/groups (e.g., `nobody`) at the end of initialization before handling user requests. In Figure 3.5, the switch is performed inside `mainInitialize`. As the checkers are invoked in the end of the initialization, the checking results are not affected by user/group switches.

To capture the TOCTTOU errors, PCHECK also supports running the generated checkers periodically in a separate thread. Periodical checking is particularly useful for catching configuration errors that occur after the initial checking (e.g., due to environment changes such as remote host failures and inadvertent file deletion).

Avoiding redundant checking. PCHECK abandons the redundant checkers which are constructed from instructions that would be executed before reaching the invocation location—any configuration errors reported by such checkers should have already been detected by the system's built-in checks, or have been exposed when the configuration value is used, before the checker is called.

Creating standalone checking programs. Another option to invoking early checkers is to create a standalone checking program comprised of the checkers, and run it when the configuration file changes. This approach eliminates the need to deal with internal side effect; on the other hand, the checking program is still prohibited to have external side effect. Note that the generated checkers start from the instructions that load configuration values (§3.4.1); therefore, the checking program needs to include the procedures that parse configuration files and store configuration values. This is straightforward for the software systems with modularized parsing procedures[7], but could be difficult if the parsing procedures cannot be easily decoupled from the initialization phase (the initialization may have external side effects).

3.5 Experimental Evaluation

3.5.1 Methodology

We first evaluate the effectiveness of PCHECK using the 37 new LC errors discovered in our study. As discussed in §3.3, all these new LC errors are from the latest versions of the systems; any of them can impair the corresponding RAS features such as fail-over and error handling.

As the design of PCHECK is inspired by the above LC errors, our evaluation contains two more sets of benchmarks to evaluate how PCHECK works beyond these errors. First, we evaluate PCHECK on a distinct set of 21 real-world LC errors that caused system failures in the past. These LC errors are collected from the datasets in prior studies related to configurations [18,56,175,185,190]; all of them were introduced by real operators and caused real-world failures. Some of these cases have different code patterns from the ones we discovered in §3.3. Table 3.7 lists the number of these LC

[7]We implement this approach for HDFS, YARN, and HBase which use modularized getter functions to parse/store configuration values.

Table 3.7. The number of LC error cases used in the evaluation, and the setup efforts (the lines of specifications for identifying starting points, c.f., §3.4.1 and annotations of invocation location, c.f., §3.4.4).

Software	Historical	New	Setup effort
HDFS	7	17	6
YARN	6	9	7
HBase	3	3	6
Apache	2	3	6
Squid	2	3	4
MySQL	1	2	31
Total	21	37	N/A

errors in each system. Furthermore, we apply PCHECK to 830 configuration files of the studied systems (except Squid) collected from the official mailing lists of these systems and online technical forums such as ServerFault and StackOverflow [1]. This simulates the experience of using PCHECK on real-world configuration files (§3.5.2). Moreover, it helps measure the false positive rate of the checking results (§3.5.6).

Note that we evaluate PCHECK upon all types of LC errors, instead of any specific error types. Therefore, the evaluation results indicate the checking effectiveness of PCHECK in terms of all possible LC errors. Table 3.8 categorizes and exemplifies the LC errors used in the evaluation based on their types.

Also, the evaluation does not use synthetic errors generated by mutation or fuzzing tools (e.g., ConfErr [83]). Most of the synthetic errors are not LC errors—they are manifested or detected by the system's built-in checks at the system's initialization time. Thus, using such errors would make the results less meaningful to LC errors.

For each system, we apply PCHECK to generate the early checkers and insert them in the system's program. Table 3.7 lists the setup efforts for the each system evaluated, measured by the lines of specifications for identifying the start points (c.f., §3.4.1) and annotations of the invocation locations (c.f., §3.4.4). Then, we apply the auto-generated checkers to the configuration files that contain these LC errors. We eval-

Table 3.8. Types and examples of LC errors used in the evaluation.

Type 1:	**Type and format errors (14 cases)**
Ex. 1:	Ill format settings, e.g., with untrimmed space [61, 63];
Ex. 2:	Invalid type settings, e.g., 0.05 for integer [60];
Type 2:	**Undefined options or ranges (6 cases)**
Ex. 1:	Deprecated compression codec class set by operators [62];
Ex. 2:	Unsupported HTTP protocol settings [64];
Type 3:	**Incorrect file-path settings (19 cases)**
Ex. 1:	Non-existent paths which will be opened or executed [142];
Ex. 2:	Wrong file types, e.g., set regular files for directories [106];
Type 4:	**Other erroneous settings (19 cases)**
Ex. 1:	Negative values used by `sleep` and thread `join` [65, 182];
Ex. 2:	Invalid mail program [143] and unreachable emails [143];

uate the effectiveness of PCHECK based on how many of the real-world LC errors can be reported by the auto-generated checkers.

We compare the checking results of PCHECK with CONF_SPELLCHECKER[8], a state-of-the-art static configuration checking tool built on top of automatic type inference of configuration values [124]. For each defined type, CONF_SPELLCHECKER implements corresponding checking functions which are invoked to check the validity of the configuration settings.

3.5.2 Detecting Real-world LC Errors

PCHECK detects 70+% of both historical and new LC errors (as shown in Table 3.9), preventing the latent manifestation and resultant system damage imposed by these errors. The results are promising, especially considering that we evaluate PCHECK using all types of configuration errors instead of any specific type. Indeed, PCHECK is by design generic to any types of configuration errors that can be exposed through execution emulation. Many of these LC errors cannot be detected by the state-of-the-art detection tools, as discussed below and in §3.5.3.

[8]https://github.com/roterdam/jchord/tree/master/conf_spellchecker

Table 3.9. The number (percentage) of the LC errors detected by the early checkers generated by PCHECK. PCHECK detects 7 (33.3%) and 11 (29.7%) more LC errors among the historical and new LC-error benchmarks respectively, compared to CONF_SPELLCHECKER, a state-of-the-art configuration-error detection tool.

Types of LC errors	**# (%) LC errors detected**			
	Historical		**New**	
Type and format error	1/1	(100.0%)	13/13	(100.0%)
Undefined option/range	2/2	(100.0%)	4/4	(100.0%)
Incorrect file/dir path	9/12	(75.0%)	5/7	(71.4%)
Other erroneous setting	3/6	(50.0%)	7/13	(53.8%)
Total	**15/21**	**(71.4%)**	**29/37**	**(78.4%)**

Among the different types of LC errors, PCHECK detects all the errors violating the types/formats and options/ranges constraints. These two types of errors usually go through straightforward code patterns and do not have dependencies with the system's runtime states. For example, most type/format errors in HDFS and YARN are manifested when these systems read and parse the erroneous settings through the getter functions. As the auto-generated checkers invoke the getter instructions, it triggers exceptions and detects the errors.

PCHECK detects the majority of LC errors that violate file-related constraints (including special files such as directories and executables). We observe that the majority of the file parameters fall into recognized APIs, such as `open`, `fopen`, and `FileInputStream`. The undetected file-related LC errors are mainly caused by (1) unknown external usage and (2) indeterminate context. The former prevents the generated checkers from being executed, and the latter stops generation of the checkers. For example, some errors reside in parameters whose values are concatenated into shell command strings, used as the argument of `system()` (to invoke `/bin/sh` to execute the command). As PCHECK has no knowledge of any shell commands, it removes the `system()` call because the side effects are unknown. The other undetected errors are in directories or file prefixes which are merged with dynamic contents from user requests which cannot be obtained statically; thereby, the corresponding checkers cannot be generated. These two causes (unknown external usage and indeterminate context) also account for the undetected errors in the "other" category.

In general, PCHECK is effective in checking errors that are manifested through execution anomalies with error identifiers defined in §3.4.3, such as those failing at system/library calls or throwing exceptions in the controlled branch. Whereas, it is hard for PCHECK to detect errors defined by application-specific semantics, such as email addresses, internal error code, etc.

We apply CONF_SPELLCHECKER on the same sets of LC errors. Compared with PCHECK, CONF_SPELLCHECKER detects 7 (33.3%) and 11 (29.7%) less LC errors in the historical and new error benchmarks, respectively. The main reason for PCHECK's outperformance is that the execution emulation can achieve fine-grained checking towards high fidelity to the original execution. For example, CONF_SPELLCHECKER can only infer the type of a configuration setting to be a "File". However, it does not understand how the system accesses the file in the execution. Thus, it reports errors if and only if "*the file is neither readable nor writable*". This heuristic would miss LC errors such as read-only files to be written by the system. Furthermore, type alone only describes a subset of constraints. CONF_SPELLCHECKER misses the LC errors that violate other types of constraints such as data ranges.

3.5.3 Checking Real-world Configuration Files

We apply the checkers generated by PCHECK to 830 real-world configuration files. PCHECK reports 282 true configuration errors and three false alarms (discussed in §3.5.6). As shown in Table 3.10, many (37.5%–87.8%) of the reported configuration errors can only be detected by considering the system's native execution environment. These configuration settings are valid in terms of format and syntax (in fact,

Table 3.10. Configuration errors detected by applying the checkers on real-world configuration files. Many of the errors can only be detected by considering the system's native environment (§3.5.3).

Software	# config files	# (%) detected configuration errors	
		All	Environment specific
HDFS	245	40	15 (37.5%)
YARN	81	49	32 (65.3%)
HBase	405	139	95 (68.3%)
Apache	65	41	36 (87.8%)
MySQL	34	13	10 (76.9%)

they are likely to be correct in the original hosts). However, they are erroneous when used on the current system because the values violate environment constraints such as undefined environment variables, non-existent file paths, unreachable IP addresses, etc. Since PCHECK emulates the execution that uses the configuration values on the system's native execution environment, it naturally detects these errors. On the other hand, such configuration errors are not likely to be detected by traditional detection methods [115,118,131,165,189] that treat configuration values as string literals, and thus are agnostic to the execution environment.

3.5.4 Checker Generation

Table 3.11 shows the number of configuration parameters that have checkers generated by PCHECK and the total number of generated checkers for the evaluated systems (multiple checkers could be generated for a parameter).

PCHECK generates checkers for every recognized parameter of HDFS, YARN, and HBase. Each emulated execution in these systems starts from the call instructions of getter functions, so the checkers are able to capture all the errors starting from the parsing phase to the usage phase. For Apache, MySQL and Squid, PCHECK generates fewer checkers. As these systems parse and assign parameter settings to corresponding program variables at the initialization stage, PCHECK bypasses the parsing phase and

Table 3.11. The number of parameters with checkers generated by PCHECK and the total number of generated checkers (each represents a distinct parameter usage scenario).

Software	# checked parameters (# checkers)	All parameters
HDFS	164 (252)	164
YARN	116 (200)	116
HBase	125 (201)	125
Apache	18 (41)	97
Squid	45 (74)	216
MySQL	32 (51)	462

directly starts from the variables that store the configuration value. Since a large number of the Boolean and numeric variables are only used for branch control with no error identifier (both branches are valid), PCHECK does not generate checkers for them (c.f., §3.4.3). Moreover, many of the variables are only used at the initialization phase before reaching the invocation location, so their checkers are considered redundant and thus are abandoned (c.f., §3.4.4).

The other issues that prevent checker generation include dependencies on the system's runtime states and uses of customized APIs (e.g., Apache uses customized `APR` string operations which heavily rely on predefined memory pools). Fortunately, as shown in §3.5.2, the majority of the LC errors have standard code patterns and can be detected using PCHECK's approach. Generating checkers for the rest of the errors require more advanced analysis and program-specific semantics.

Also, we can see that the total number of checkers are well bounded, which is attributable to the execution merging (§3.4.1) and redundancy elimination (§3.4.4).

3.5.5 Checking Overhead

The checkers are only invoked at the initialization phase or run in a separate thread, thus they have little impact on the systems' runtime performance. We measure their overhead to be the time needed to execute these checkers, by inserting time coun-

Table 3.12. Checking overhead (time needed to run the auto-generated checkers).

Software	**Time for running the checkers (millisec.)**			
HDFS	[NameNode]	408	[DataNode]	311
YARN	[ResourceMgr]	243	[NodeMgr]	486
HBase	[HMaster]	780	[RegionServer]	777
Apache	[httpd]	0.6	————	—
Squid	[squid]	93.8	————	—
MySQL	[mysqld]	1.7	————	—

ters before and after invoking all the checkers. Table 3.12 shows the time in milliseconds (ms) to run the checkers on a 4-core, 2.50GHz processor connected to a local network (for distributed systems like HDFS, YARN, and HBase, the peer nodes are located in the same local network). The checking overhead for Apache and MySQL is negligible (less than 5ms); Squid needs around 100ms because it has a parameter that points to public IP addresses (`announce_host`). The overhead for the three Java programs is less than a second. The main portion of the time is spent on network- and file-related checking. Since PCHECK only performs lightweight checks (e.g., metadata checks and reachability checks), the overhead is small. Note that the checkers are currently executed sequentially. It is straightforward to invoke multiple checkers in parallel to reduce overhead, as all the checkers are independent.

3.5.6 False Positives

We measure false positives by applying the checkers generated by PCHECK to both the default configuration values of the evaluated systems and the 830 real-world configuration files, and examine whether or not our checkers would falsely report errors. We also manually inspect the code of the generated checkers in LLVM IR and Jimple to look for potential incorrectness.

Among all the configuration parameters in the evaluated systems, only three of them have false alarms reported by the auto-generated checkers: two from YARN and one from HBase. All these false positives are caused by the checkers incorrectly skipping conditional instructions affected by the configuration value (§3.4.1), due to unsound static analysis that misses control dependencies. This results in emulating the execution that should never happen in reality—certainly, the anomalies exposed in such execution are unreal. The overall false positive rates are low. YARN has the most configuration parameters with false checkers, with the false positive rate of 1.7% (2 over 116 parameters). Note that checkers with false positives can be removed by the developers or disabled by the operators in the field (c.f., §3.4: Usage).

3.6 Limitations

No tool is perfect. PCHECK is no exception. Like many other detection tools for bugs and misconfigurations, PCHECK is neither sound nor complete for its checking scope and the design trade-offs.

PCHECK targets on the scope of configuration errors manifested through explicit, recognizable instruction-level anomalies (c.f., §3.4.3). It cannot detect *legal* misconfigurations [185] that have valid values but do not deliver the intended system behavior. The common legal misconfigurations include inappropriate configuration settings that violate resource constraints or performance requirements (e.g., insufficient heap size and too small timeout). Such misconfigurations are notoriously hard to detect and are often manifested in a latent fashion as well, such as runtime out-of-memory errors [49] (resources are not used up immediately). However, detecting resource- and performance-related misconfigurations would need dynamic information regarding resource usage and performance monitoring [68, 161], which is beyond the static methods of PCHECK.

In addition, PCHECK cannot emulate the execution that depends on runtime inputs/workloads, or does not have statically determinate context in the program code (c.f., §3.4.1). Thus, it would miss the configuration errors that are only manifested during such execution. Nevertheless, indeterminate context (e.g., those derived from inputs and workloads) can potentially be modeled with representative values, which could significantly improve the capability of checker generation.

One design choice we make is to trade soundness for safety and efficiency—PCHECK aims to detect common LC errors without incurring side effects or much overhead. For example, PCHECK does not look into file contents but only checks if the file can be accessed as expected. Similarly, PCHECK only checks the reachability of a configured IP address or host instead of connecting and sending packets to the remote host. It is possible that certain sophisticated errors can escape from PCHECK (e.g., the configured file is corrupted and thus has wrong contents). As the first step, we target on basic, common errors, as they already account for a large number of real-world LC errors [97,155,185]. Efficiently detecting sophisticated errors may require not only deeper analysis but also application semantics.

3.7 Summary

This chapter advocates early detection of configuration errors to minimize failure damage, especially in cloud and data-center systems. Despite all the efforts of validation, review, and testing, configuration errors (even those obvious errors) still cause many high-impact incidents of today's Internet and cloud systems. We believe that this is partly due to the lack of automatic solutions for cloud and data-center systems to detect and defend against configuration errors (the existing solutions are hard to be applied, due to their strong reliance on datasets). We envisage that PCHECK is the first step towards a generic and systematic solution to detect configuration errors. PCHECK does not require collecting any external datasets and is not specific to any specific rules. It detects configuration errors based on how the system actually uses the configuration values. With PCHECK, we demonstrate that such detection method can effectively detect real-world LC errors, with little runtime overhead and setup effort. The original paper [176] has been used to support proactive configuration checking in existing software systems.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, 2016. Xu, Tianyin; Jin, Xinxin; Huang, Peng; Zhou, Yuanyuan; Lu, Shan; Jin, Long; Pasupathy, Shankar. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Simplicity-oriented Design

“*Everything should be made as simple as possible, but no simpler.*”
—Albert Einstein

While Chapter 2 and 3 provide practical solutions that enable existing software systems to defend against configuration errors, the ultimate solution is re-thinking and re-designing configuration to prevent (or at least significantly reduce) potential misconfigurations in the first place. One fundamental reason for today’s prevalent configuration issues (including misconfigurations) is the tremendous and still increasing complexity of configuration, as revealed by the previous studies [27, 38, 134, 180] (we will discuss the configuration complexity special for systems software in §4.2). This chapter explores the feasibility and efficacy of simplifying configuration design, as an attempt to make software configuration more usable and less error-prone.

4.1 Introduction

The key towards usable configuration is to understand the difficulties and the resulting mistakes/errors from the perspective of users[1]—human operators, as many misconfigurations are essentially introduced by human mistakes and errors [178]. Therefore,

[1]In this chapter, we sometimes use “user” instead of “operators” to emphasize the perspective of treating configuration as an interface and treating operators as the *users*.

before jumping onto the simplification track of configurations, we strive to understand a fundamental question: "*Do operators really need so many configuration 'knobs', and how do they configure the systems in the field?* The chapter starts with a quantitative study of real-world configuration usage characteristics based on the configuration settings collected from many thousands of real operators of a commercial system from a major storage company in the U.S., and also the settings of hundreds of operators of two widely-used open-source server software projects. The study also includes the analysis of 620 user-reported configuration issues (from both the commercial and open-source software projects), in order to understand the configuration problems caused by the complexity of current configuration design.

The study provides quantitative evidence that we (software developers) are providing more configurations than what the majority of operators need or know how to set. Many configuration knobs are neither necessary nor worthwhile—they make configuration more complex for common operators, but produce little benefit as flexibility desired by operators. Unfortunately, complexity does come with a cost. It prevents operators from understanding every configuration thoroughly and examining its settings carefully. For example, a significant percentage of user-reported configuration problems are about their difficulties in finding or setting the correct configurations to achieve the desired system behavior; also, many operators' incorrectly keeping the default values for parameters that need to be set based on the runtime environments.

These findings drive us to tackle the over-designed configuration complexity. First, we propose a number of concrete, practical design guidelines which could significantly reduce the configuration space of existing systems software, thus simplifying the inherent complexity of configuration design. We show that these guidelines incur little impact on the flexibility desired by the operators. Second, we measure the efficacy of existing configuration navigation solutions, including COX [2] (our solution that helps operators find the right knobs by expressing intent, based on natural language processing). Based on this, we provide practices for building tool support to help operators navigate the vast configuration space in existing software systems.

4.2 Background

One fundamental reason for today's prevalent configuration issues is the ever-increasing complexity of configuration, especially in systems software. This is reflected by the large and still increasing number of configuration parameters ("knobs"), as well as various configuration constraints and consistency requirements [85, 107, 124, 179] (known as complexity of interaction and tightness of coupling in human error studies [117, 127]). For example, MySQL 5.6 database server has 461 configuration parameters; 216 of them are not with simple data types (e.g., Boolean or enumerative) but rather more complex ones. These parameters control different buffer sizes, timeouts, resource limits, etc. Setting them correctly requires domain-specific knowledge and experience. Other server software is similar. For example, Apache HTTP server 2.4 has more than 550 parameters across all the modules. Moreover, many of these parameters have dependencies and correlations [126, 190], which further worsens the situation. Such high complexity level makes system configuration a daunting and error-prone task, with prevalent misconfigurations being the after-effects.

Figure 4.1 depicts how the number of configuration parameters increases with software evolution of one commercial storage system[2] and three popular open-source server software projects. Take Hadoop MapReduce as an example, its first release in Apr. 2006 had only 17 parameters, but the release in Oct. 2013 already had 173, an increase of more than nine times. In accordance with this trend, the configuration space will continue to increase. The situation is further worsened by the growing number of

[2]We are required to keep the company and the products anonymous.

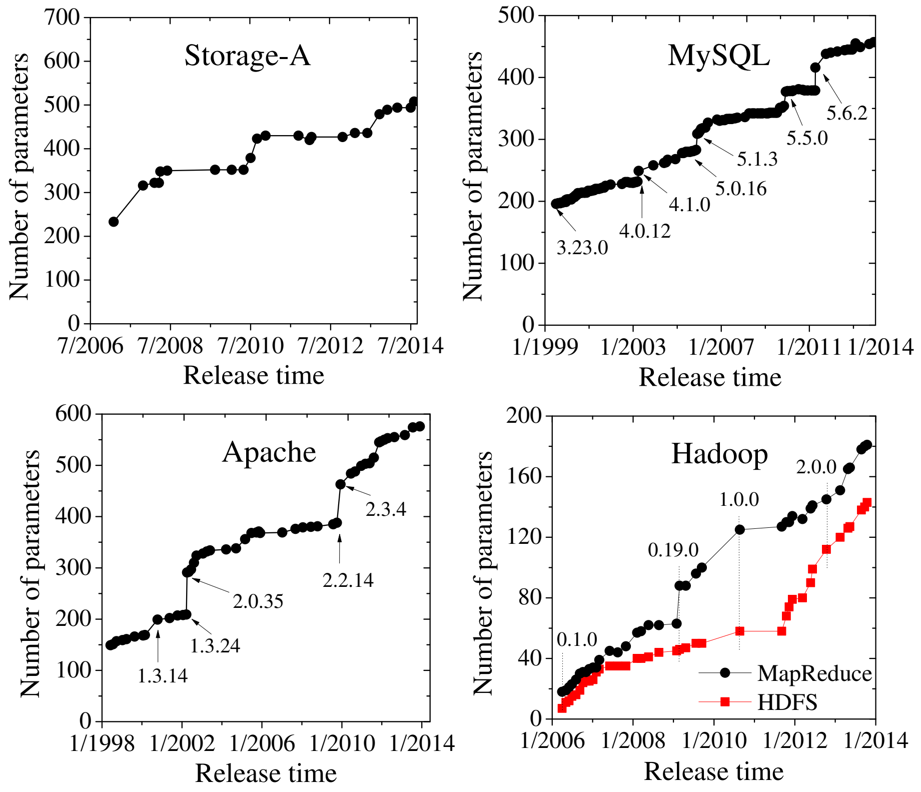
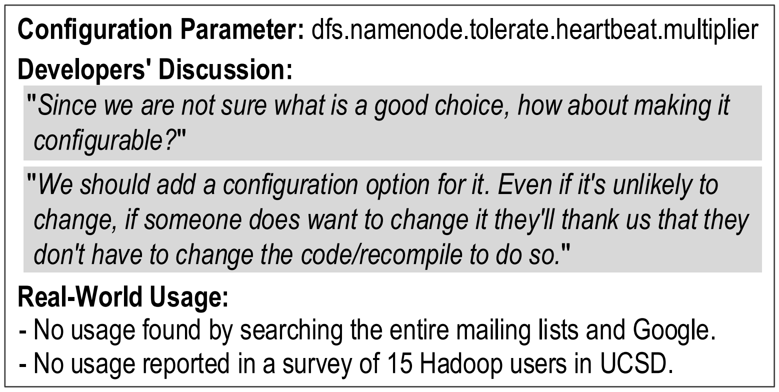


Figure 4.1. The increasing number of configuration parameters with software evolution. Storage-A is a commercial storage system from a major storage company in the U.S.

machines that replicate multiple software instances in data centers, and the increasing cost of human resources for managing them.

Typically, a configuration parameter is introduced when the developers want to provide flexibility for operators. As one type of system interfaces, configuration needs to be balanced between flexibility and simplicity. This raises the question if it is worth satisfying a few advanced operators or even imaginary operators, while confusing the majority of ordinary operators. Often, we (software developers) seem to be biased towards "advanced" operators instead of focusing on the ease of use. Figure 4.2 shows an example where the Hadoop developers exposed a parameter in case some advanced operators want it. However, since the parameter is specific to the internal system implementation, few operator has set it.



Configuration Parameter: dfs.namenode.tolerate.heartbeat.multiplier

Developers' Discussion:

"*Since we are not sure what is a good choice, how about making it configurable?*"

"*We should add a configuration option for it. Even if it's unlikely to change, if someone does want to change it they'll thank us that they don't have to change the code/recompile to do so.*"

Real-World Usage:

- No usage found by searching the entire mailing lists and Google.
- No usage reported in a survey of 15 Hadoop users in UCSD.

Figure 4.2. A real-world example of less useful configuration parameters from HDFS. This parameter is specific to internal system implementation and seldom set by any operator. Such parameters should not be exposed to operators (at least not to the common operators).

Table 4.1. The average number of added, renamed, and removed configuration parameters per version release. The numbers are obtained from the official user manuals.

Software	Addition	Renaming	Removal
Storage-A	13.65	0.26	1.87
Apache	5.19	0.23	0.61
MySQL	2.54	0.06	0.53
Hadoop	4.14	1.60	0.39

It is worth noting that many configuration parameters are added with new software versions released, but are removed at a much slower rate. The slow rate is probably due to backward compatibility concerns, or developers' lack of sufficient knowledge or confidence in removing parameters introduced by someone else. Table 4.1 shows the number of parameter changes during software evolution. The parameter removal rate is almost 7x slower than the rate of addition (Storage-A has a faster rate than the open-source software because its release cycles are longer). Following such trends, we will be presented with more configuration parameters in the future releases, unless we take serious actions to simplify them.

Many desktop and mobile applications adopt GUI-based methods such as preference menus [79] to reduce user-perceived complexity caused by large preference space

Table 4.2. The target systems software in the study

Software	Proprietary	Language	Dev. history	# parameters
Storage-A	Commercial	—	22 years	412
Apache	Open source	C	21 years	587
MySQL	Open source	C++	21 years	461
Hadoop	Open source	Java	9 years	312

(e.g., colors, fonts, and layouts). Unfortunately, due to the scalability and accessibility issues [162], the primary and *de facto* configuration interfaces of systems software are text files (e.g., in xml and .ini formats). GUIs are not widely used for systems software configuration [19, 58, 81, 153, 162], making GUI-based methods hard to apply.

In fact, some software developers have already sensed that today's configuration is overly complex and should be simplified to some degree. For example, Rob Pike, the developer of Unix and the Go programming language recently commented, "*There is too much configuration. There are too many options. There are too many dot files. Stuff should just work.*" This chapter intends to systematically understanding and dealing with the over-designed configuration in existing systems software.

4.3 Methodology

4.3.1 Target Software

We study four systems software projects, including the software of one commercial storage system and three open-source systems software, as shown in Table 4.2. The commercial system, Storage-A, is from a major storage company in the U.S. It runs distributed storage software to manage network-attached storage devices. The open-source software includes Apache HTTP server, MySQL data- base server, and Hadoop (including MapReduce and HDFS). All these software projects are mature (with 9$\sim$21 years of development history) and widely used, representing different types of systems software (storage, Web, database, and data processing).

Table 4.3. Configuration setting datasets used in this study. Note: The numbers of parameters for Apache and MySQL are different from those in Table 4.2, as we only study the common parameters of the selected versions and exclude parameters of specific OS/modules (§4.3.2).

Software	Version	# parameters	System instances
Storage-A	a.b.c	412	many thousands
Apache	2.2.x	90	168
MySQL	5.x	221	260

Note that all the studied software projects fall into the category of *systems software*, which is used to provide services for client-side application software (e.g., Web browser, file manager). The users of systems software are mostly system operators who usually have higher level of technical background and skills than ordinary end users. This is particularly true for the commercial storage systems which are mainly purchased and used by enterprise customers (instead of individuals).

4.3.2 Real-world Configuration Settings

We collect configuration settings of real operators of Storage-A, Apache, and MySQL. The Hadoop dataset we collected is not large enough to be statistically significant. Thus, we exclude Hadoop from the first part of our study on configuration settings. Table 4.3 summarizes the configuration setting datasets.

The dataset of Storage-A includes the configuration settings of *all the customers using the same version of Storage-A* (anonymized as "a.b.c"). It contains many thousands of customers' settings spanning one and a half years (from Jun. 1, 2012 to Dec. 31, 2013). The data was collected by a support system which recorded the customers' configuration settings on a weekly basis. In our study, we use each customer's most recent configuration settings. Note that this dataset provides the *exhaustive ground truth* of the configuration settings of Storage-A, version-a.b.c in the field, and is used as the primary data source in our study.

As the complimentary data references to verify the findings discovered from the Storage-A dataset, real-world configuration settings for Apache and MySQL are also collected from the Internet. We crawl configuration files attached by users from well-known online forums (including ServerFault[3], StackOverflow[4], Webmasters[5], and Database Administrators[6]), and the entire archives of the official mailing lists of the two software projects. We only collect complete configuration files attached by users who posted the questions, and exclude any partial configuration snippets included in postings, because partial snippets does not reflect all the settings made by operator (we cannot know the settings of the parameters not occurring in the snippets); including them would cause bias to the appeared parameters. Moreover, we only collect *one configuration file per user* (identified by user IDs/emails) to ensure the representativeness of our datasets.

We use the configurations of the same major version for each software project so that we do not need to deal with the difference across versions. To make sure the studied parameters were presented to all the operators, we exclude parameters designed for specific plugins or OS (e.g., OS/2, BeOS) that are not the default OS and software modules of the studied software and are not needed for the majority of the users.

4.3.3 Real-world Configuration Issues

We collect 620 real-world cases of user-reported configuration issues related to the studied systems software. Table 4.4 shows the numbers of collected cases of each software project. The cases of Storage-A are collected from the commercial company's customer-issue database which records the issues reported by the customers. We randomly sampled 1,000 cases labeled as "configuration related" by technical support engineers, and only selected those that have been resolved and confirmed by the original

[3]http://serverfault.com/

[4]http://stackoverflow.com/

[5]http://webmasters.stackexchange.com/

[6]http://dba.stackexchange.com/

Table 4.4. Real-world configuration-related issues included in the study.

Software	**Studied cases**
Storage-A	329
Apache	97
MySQL	96
Hadoop	98

users. For the open-source software, we collect user-reported configuration issues from two sources: the software's official mailing lists and the well-known online forums (the same as in §4.3.2). This study focuses on issues related to parameter configuration, as they account for the majority of real-world configuration problems [185].

4.3.4 Threats to Validity

As with all characteristic studies, there is an inherent risk that our findings may be specific to the studied software and may not apply to other software. While we cannot establish representativeness categorically, we have taken care to select diverse software with different proprietary licenses, functionalities, and languages of implementation (c.f., Table 4.2). As all the studied software projects are mature and widely used, we believe that their configuration accurately represents the configuration design practices of today's systems software. However, our study only focuses on systems software; we do not intend to draw any conclusion about other types of software, such as desktop software and mobile applications.

Another potential source of bias is in the collection of configuration settings of the open-source software. The configuration files of Apache and MySQL are collected from online forums and mailing lists. Many of them are associated with configuration problems encountered by operators; some of these files may have a few erroneous settings. Although these settings (including the erroneous ones) are configured and applied by real operators, we acknowledge that the datasets may be biased to operators who encountered configuration problems. To avoid the impact of the potential biases, we only use the datasets as complimentary references to the Storage-A dataset (which does not have such bias). We do not draw any conclusion directly from the open-source datasets. All the reported findings are discovered in Storage-A and then verified in the open-source datasets. As described in §4.3.2, *the Storage-A dataset contains all the settings of Storage-A customers of the same version, which is exhaustive without bias to any special type of operators.*

Another concern is the representativeness of configuration issues. We collect only user-reported issues. It is possible that operators do not report easy-to-solve problems. Also, novice operators are more likely to report problems, compared with experts. Unfortunately, it is hard to objectively judge whether an operator is a novice or an expert. In fact, with new or major revisions of software being deployed in the field, there are always novice operators. Therefore, our findings are still valid. Note: Our study mainly focuses on the configuration difficulties and mistakes caused by existing configuration design, instead of general characteristics of configuration errors.

Finally, we still remind readers to interpret our findings and results in the context of our studied software and datasets.

4.4 Understanding Configuration Settings in the Field

In this section, we first study how the configuration parameters are set by real operators. Then, we examine how operators handle the increasing configuration complexity. Note that *we only study the configuration parameters exposed to operators intentionally, instead of those to developers or technical-support engineers*. All the studied parameters are documented in the official user manuals, we exclude hidden parameters that are not visible to the common operators. Therefore, all the reported findings in this section are only applicable to configuration design for the operator's interface.

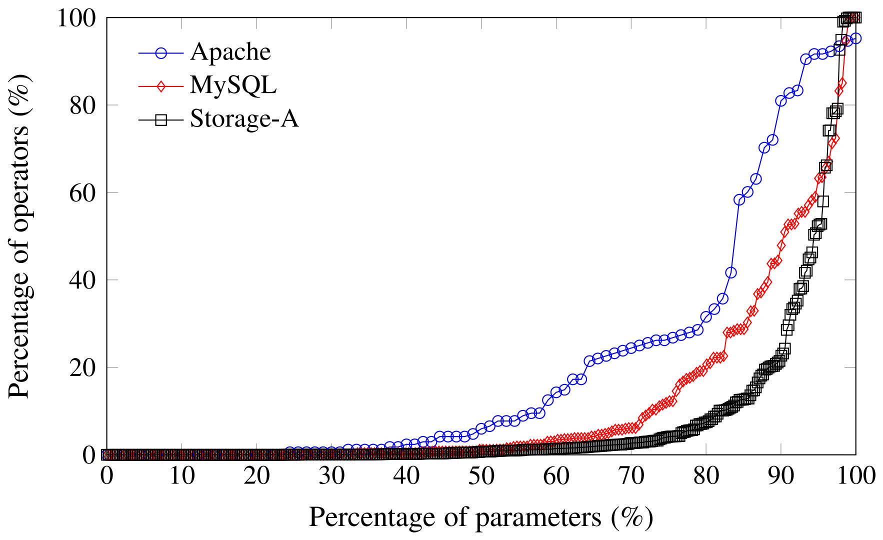


Figure 4.3. How many parameters are used in the field by the operators? Each data point (x, y) on a curve indicates that "x% of the parameters were set by fewer than y% of the operators." Table 4.5 and 4.6 further zoom into the parameters set by 0%/1-% of operators and 50+%/90+% of operators.

4.4.1 Do Operators Really Need So Many Configuration Knobs?

Finding 1(a): *Only a small percentage (6.1%∼16.7%) of configuration parameters are set by the majority of operators; a significant percentage (up to 54.1%) of parameters are rarely set by any operator.* It seems that many parameters (at least those rarely-set ones) are not necessary to most of the operators. They enlarge the configuration space (adding more complexity) without producing much benefit (in terms of the desired flexibility) to the common operators. We discuss the problems of "too many knobs" in §4.4.3. The rarely-set parameters should be separated from the commonly used ones.

Figure 4.3 plots the real-world usages of configuration parameters, measured by the percentage of operators whose settings are different from the defaults in the studied systems. Table 4.5 shows the percentage of the parameters that are seldom set (by fewer than 1% of operators); Table 4.6 gives the percentage of the parameters set by the majority (more than 50%) of operators. These results give quantitative evidence that we

Table 4.5. The percentage (number) of parameters that were set by 0% and by fewer than 1% of the operators, respectively.

Software	**% (#) of parameters**		**Total**
	% of operators = 0%	**% of operators < 1%**	**(#)**
Storage-A	23.3% (96)	54.1% (223)	412
Apache	23.3% (21)	31.1% (28)	90
MySQL	33.0% (73)	49.8% (110)	221

Table 4.6. The percentage (number) of parameters that were set by more than 50% and 90% of the operators, respectively.

Software	**% (#) of parameters**		**Total**
	% of operators > 50%	**% of operators > 90%**	**(#)**
Storage-A	6.1% (25)	2.4% (10)	412
Apache	16.7% (15)	7.8% (7)	90
MySQL	10.0% (22)	1.8% (4)	221

(software developers) seem to have provided more configuration knobs than what the majority of operators need or know how to use. The configuration parameters, rarely set by operators, should be either hidden (informing operators by requests) or removed from common operators, to avoid blowing up the user-perceived configuration space.

One may argue that operators do not set these parameters only because the default settings are good. First, this may not always be true. As we will show in §4.4.3, many times operators do not change the default settings because they do not understand the meaning or impact of the parameters and thereby do not know how to set them appropriately. Second, even the above statement is true, if almost all the operators never need to set the parameters to be different from the defaults, what is the need to expose these knobs to operators? We can keep them hidden or completely remove them so that operators can focus on the parameters that need to be changed.

Finding 1(b): *A small percentage (1.8%∼7.8%) of parameters are configured by more than 90% of the operators.* Many of these parameters provide necessary information of the system runtime which is hard to have default values in advance.

Table 4.7. The percentage of numeric parameters with no more than five distinct settings used by 90%/100% of the operators. We exclude operators who did not set the parameters.

Software	**% of parameters with five distinct values**	
	Covering 90% operators	**Covering 100% operators**
Storage-A	65.8%	47.4%
Apache	60.0%	10.0%
MySQL	26.7%	12.2%

These parameters should be exposed or recommended to the operators as the "first-class" knobs. Software developers should provide simple tutorials, guidebooks, and templates, to explain in more details about these parameters. This can help operators to focus and have an easier, quicker start, instead of skimming through a thick manual [e.g., MySQL's reference manual (version 5.6) has a length of 3,989 pages[7]].

4.4.2 Should We Offer More Choices in Configuration Knobs?

Finding 2(a): *Software developers often choose more "flexible" data types for configuration parameters to give operators more flexibility of settings (e.g., using numeric types instead of the simple Boolean or enumerative ones). However, operators seem not to take full advantage of such flexibility. A significant percentage (up to 47.4%) of numeric parameters have no more than five distinct values among all the settings.*

Once again, this implies that the developers' goodwill in providing operator with flexibility is not fully appreciated by them. Reducing the *value space* of these parameters can simplify their settings, without sacrificing much flexibility. Developers can convert the complex types into Boolean or enumerative types which are more expressive and easier for operators to configure.

Table 4.7 shows the similarity of operators' settings for numeric parameters in the studied systems. For each numeric parameter, we select the five most popular val-

[7]http://downloads.mysql.com/docs/refman-5.6-en.pdf

ues, and measure their coverage among all the settings. We exclude operators staying with default values (they did not set these parameters), in order to avoid the dominating coverage of the default values (including the default values would make the percentages even higher, because most operators go with the defaults for many numeric parameters).

Note that the majority numeric parameters have a large *range*. Many parameters do not have specified min/max values, so their ranges depend on their data types. For example, the data range of parameters represented by `unsigned int` is [0, `UINT_MAX`]. We do not normalize the results in Table 4.7 by the numeric ranges (as denominators), because large ranges (e.g., [0, `INT_MAX`] and [0, `LONG_MAX`]) seldom have meanings to operators and thus do not impact their settings (making normalization nonsensical).

One reason of the small number of settings is the distribution of templates among the user communities. For example, the communities of the configuration management tools (e.g., Puppet and Chef) have the tradition of sharing *recipes* for a variety of common software. The results indicate that the majority of operators do not need large value space for configuration parameters. Providing operators with a few representative options covering typical usage scenarios is simpler and more efficient.

Finding 2(b): *For enumerative parameters with many options, typically only two to three of the options are actually used by the operators, indicating once again the over-designed flexibility.* Figure 4.4 shows the number of used options among all the options.

Compared with numeric ones, enumerative parameters have less options with more representative values. However, if there are too many options with ambiguous semantics, operators tend to stay on a few *safe* options. For example, the `LogLevel` parameter in Apache has 16 options, corresponding to 16 different logging verbosity levels.[8] Though the user manual explains each level using log examples, only six options

[8]http://httpd.apache.org/docs/2.4/mod/core.html#loglevel

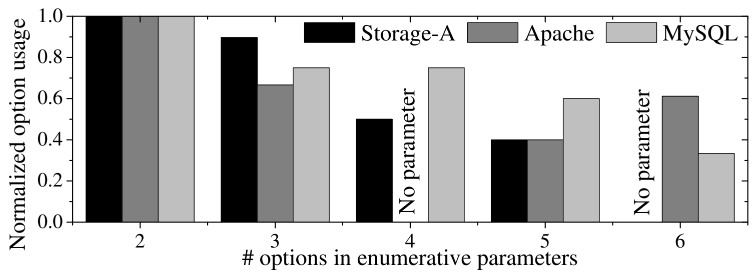


Figure 4.4. Usage of enumerative parameters with different number of options (percentages of used options among all the provided options)

appear in our datasets. The log levels for specific debugging purposes (e.g., `trace1-8`) should not be exposed to the common operators. In fact, even the developers themselves are often confused by the verbosity levels, not to mention the operators [188].

4.4.3 What Is The "Cost" of Too Many Knobs?

Some software developers may argue that most of the parameters have default values; also, operators can learn about these parameters by referring to user manuals. Thus, there is no real harm in introducing a large number of configuration parameters or providing many options for them. However, this argument is somewhat refuted by our study of real-world configuration issues reported by operators.

Finding 3: *Too many knobs do come with a cost: operators encounter tremendous difficulties in knowing which configuration parameters should be set among the large configuration space. This is reflected by the following two facts:*

- *a significant percentage (up to 48.5%) of configuration issues are about the difficulties in finding or setting the parameters to obtain the intended system behavior;*
- *a significant percentage (up to 53.3%) of configuration errors are introduced due to operators' staying with default values incorrectly.*

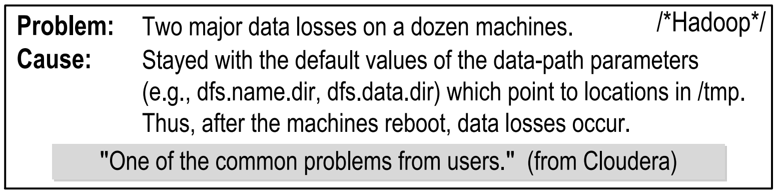
Table 4.8. The distribution of user-reported configuration issues across the categories.

Software	Difficulties	Errors	Others	Total
Storage-A	17.3% (57)	70.5% (232)	12.2% (40)	329
Apache	48.5% (47)	44.3% (43)	7.2% (7)	97
MySQL	34.4% (33)	47.9% (46)	17.7% (17)	96
Hadoop	35.7% (35)	42.9% (42)	21.4% (21)	98

To understand configuration problems faced by operators in the real world, we categorize the user-reported issues into "difficulties," "errors," and "others." The *difficulties* refer to cases where operators do not know *what* or *how* to configure to obtain their intended system functionalities or performance goals. The *errors* refer to erroneous settings that caused system misbehavior, such as crashes, hangs, or performance degradation. The operators failed to reason out the misconfigurations as the root causes, and thus called support engineers or posted the problems on online forums and mailing lists. There are other configuration-related issues such as inquiries about general practices and internal usages, categorized as "others."

Table 4.8 shows the distribution of the collected configuration issues across the categories. Remarkably, a significant percentage (17.3%$\sim$48.5%) of issues fit into the "difficulties" category. This indicates that operators face tremendous difficulties to find the right knobs from the large configuration space. It is totally understandable, given the large quantity of parameters as well as the inefficiency of common navigation practices (discussed in §4.6). Compared with open-source software, Storage-A has a lower percentage of "difficulties" issues (probably because Storage-A operators are mostly professional personnel with better configuration experience). However, 17.3% still means a large financial cost, considering the human cost of configuration-related support calls [156, 185].

Similarly, "too many knobs" may prevent operators from understanding the parameters thoroughly and tuning them carefully. Operators tend to keep the settings (e.g.,


Problem: Two major data losses on a dozen machines. /*Hadoop*/
Cause: Stayed with the default values of the data-path parameters (e.g., dfs.name.dir, dfs.data.dir) which point to locations in /tmp. Thus, after the machines reboot, data losses occur.

"One of the common problems from users." (from Cloudera)

Figure 4.5. A real-world example of configuration errors caused by the operators' incorrectly staying with default values.

Table 4.9. The number of error cases caused by the operators' incorrectly staying with default parameter values, and their percentages among all the error cases. Most of these parameters were set by more than 5% of operators. We exclude the cases in which operators did not their settings; the last column only includes parameters in our dataset.

Software	**Total (#)**	**Incorrect defaults**	**Set by <5% operators**
Storage-A	207	45.4% (94)	3.2%
Apache	40	17.5% (7)	0.0%
MySQL	45	53.3% (24)	0.0%
Hadoop	40	30.0% (12)	N/A

the default values) that work for the first run, instead of carefully, thoroughly examining the setting of every parameter. As a result, they may incorrectly miss parameters that need to be set according to the runtime environments, thus violating constraints of workloads, resources, cross-component correlations, etc. The consequence could be severe, such as failures and data losses. Figure 4.5 gives an example where the operator's incorrect staying with default values led to major data losses.

In fact, such cases (as the one in Figure 4.5) are not rare. As shown in Table 4.9, a significant percentage (17.5%∼53.3%) of the configuration errors were caused by operators' incorrectly staying with the default values, rather than setting wrong values.[9] We manually examined the operators' settings reported in the issues (the cases without enough information about operators' settings are excluded). Note that very few of these parameters are those rarely-set ones.

[9]We acknowledge the possibility of operators' intentionally setting the default values wrongly, but we believe that it is not the common case.

Parameter:	optimizer_prune_level (Boolean)	/*MySQL*/
Desc.:	Controls the heuristics applied during query optimization to prune less-promising partial plans from the optimizer search space.	
Values:	0 or 1	
Usage:	No user set the parameter in our dataset.	

(a) Empirical, heuristic usages

Parameter:	key_cache_block_size (Numeric)	/*MySQL*/
Desc.:	The size in bytes of blocks in the key cache.	
Values:	[512, 16384]	
Usage:	All the users stay with the default value 1024 in our dataset.	

(b) Control internal data structures

Figure 4.6. Two examples of configuration parameters that are seldom set by any operator in the MySQL dataset.

We cannot draw conclusions that smaller configuration space will definitely reduce the error-proneness of configuration activities, as the current datasets do not allow us to study the correlation between the size of the configuration space and the number (or rate) of configuration errors. However, as reducing the configuration space surely simplifies the configuration process, we believe it to have positive effect on the error-proneness of configuration.

4.4.4 What Kinds of Knobs Are Most Utilized?

Finding 4: *Configuration parameters with explicit semantics, visible external impact are set by more operators, in comparison to parameters that are specific to internal system implementation.* Thus, software developers should avoid exposing configuration parameters specific to internal implementation—operators cannot or may not have time to read the source code.

The distinct usages of different parameters drive us to think about the rationale behind how operators set configuration parameters. To understand the rationales behind the distinct usage of different configuration parameters, we comparatively examine

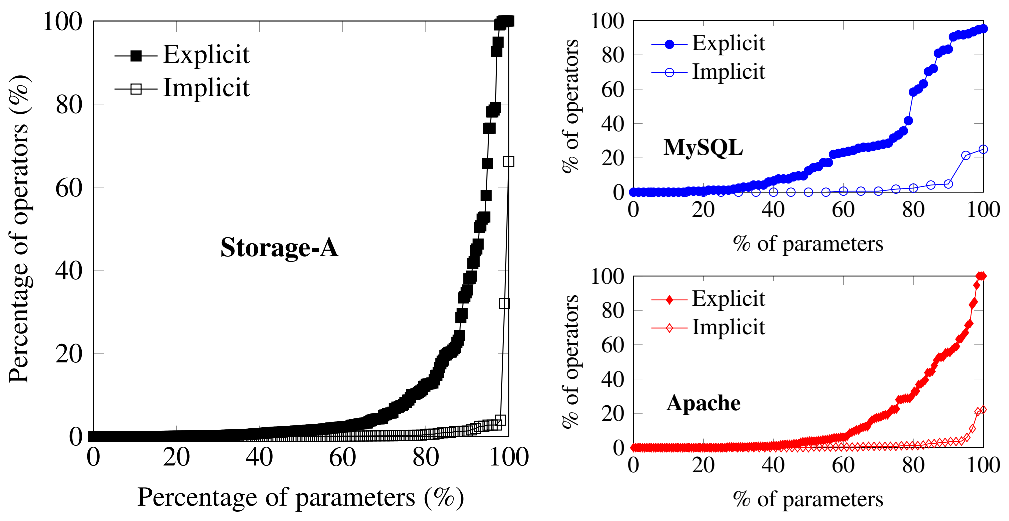


Figure 4.7. Real-world usages of configuration parameters with explicit, visible external impact (“explicit”) versus parameters specific to internal implementation (“internal”)

the parameters set by the majority of operators and those seldom set. This comparison reveals remarkable differences among these parameters. Most of the frequently-set parameters have explicit semantics or visible external impact, e.g., enabling functionalities, switching between policies, enabling backup services. Thus, it is easy for operators to understand and observe the effect of their settings. On the contrary, many seldom-set parameters are *specific* to internal system implementation or protocol details (e.g., controlling data structures or library/system calls) and empirical/heuristic usages. Figure 4.6 gives two examples of such knobs from MySQL.

Since most operators have limited knowledge about system internals (even for the open-source ones), it is difficult for them to understand the semantics and potential impact of those internal parameters. As a result, most operators do not have the confidence to touch these parameters, especially for systems software running in production systems whose availability and performance are critical. To validate our hypothesis, we manually annotate every studied parameter as “internal” or “explicit,” based on whether or not it controls internal implementation specific to the software. To minimize the subjectiveness during annotation, two inspectors separately labeled the parameters and compared the results with each other before consensus was reached. For some tough cases, we consulted with the developers to make decisions on the labeling. Figure 4.7 shows the usages of "explicit" and "internal" parameters in the studied software. It confirms that the configuration parameters specific to internal implementation are seldom set by operators.

Note that the usages of parameters are not strongly correlated with their data types. For example, Boolean parameters are usually more simple to set, compared with numeric parameters. However, if they are specific to internal implementation, they are still seldom set by operators, as exemplified in Figure 4.6a.

4.5 Configuration Simplification

We study the opportunity and effectiveness of simplifying configuration by reducing the parameter and value space, as the fundamental approach to dealing with "too many knobs." We discuss other aspects of configuration simplification in §4.7.

4.5.1 Simplification Guidelines

The findings in §4.4 lead to a set of concrete, practical guidelines for simplifying configurations. Table 4.10 summarizes these guidelines, which mainly include the following two aspects:

- **Vertical.** Hiding or removing unnecessary configuration parameters and promoting the important ones (which are usually a small set), so that operators can efficiently, correctly find the knobs.
- **Horizontal.** Reducing the value space of the parameters and providing meaningful, expressive options to help operators set the parameters correctly and efficiently.

Table 4.10. Guidelines for simplifying configuration design.

Guideline	Support	Ref.
1. Hide or remove configuration parameters that are seldom set by any operator. This requires building operator-feedback loops for configuration settings.	Finding 1(a)	§4.4.1
2. Promote parameters set by most operators to be the "first-class" ones. Include them in tutorials/guidebooks to let operators focus on these parameters first.	Finding 1(b)	§4.4.1
3. If possible, convert numeric parameters into enumerative or Boolean types with expressive, representative values to make the settings simple.	Finding 2(a)	§4.4.2
4. Avoid enumerative parameters with too many options. Typically, five options should be sufficient in terms of operator flexibility.	Finding 2(b)	§4.4.2
5. Only expose the configuration parameters with explicit semantics and/or visible external system impact.	Finding 4	§4.4.4

Note: These guidelines are only applicable to configuration design for the operators of the software, not for developers or support engineers. For example, hiding parameters should not prevent test engineers from finding or setting them. We discuss the implications of simplification to testing and debugging in §4.7.

The proposed guidelines in Table 4.10 are general and do not consider system- or domain-specific information (which may provide opportunities to further simplify configuration). In this study, we judge the necessity of a configuration parameter based on its setting statistics in the field. It is absolutely possible that even the configuration parameters set by many operators can be eliminated, e.g., by automatically inferring or generating values from runtime environments (e.g., [13,37,45,194]), formal models and specifications (e.g., [100,133,154,166]), historical settings (e.g., [195]), etc. We discuss the other aspects of configuration simplification in §4.7.2

Software vendors may raise the concern that simplifying configuration would hurt the advanced operators who do need more flexibility compared with ordinary operators; specially, some parameters are still under use even though by few operators (e.g., 1%). In fact, this problem can be gracefully addressed by decoupling the advanced configuration from the basic ones. For example, the advanced parameters can be hidden from the common operators and only be informed by requests; arbitrary parameter values are still allowed to be set if the operator insists. Nevertheless, such advanced configuration should be introduced to operators in separate manuals, templates, and files.

Please note that *we do not mean to prevent operators from fine-tuning the configuration.* Instead, we advocate better design to facilitate operators' configuration tuning by making it simple and less prone to errors. As we have demonstrated, the current design of configuration clearly does not consider operators in the design process but assume that they need and can handle the level of complexity.

4.5.2 Effectiveness of Simplification

Finding 5: *The configuration of the studied software can be significantly simplified by reducing the configuration space both vertically and horizontally. For Storage-A, 51.9% of the original configuration parameters can be hidden or removed, and 19.7% of the remaining ones can be further converted into simpler types, with the impact on fewer than 1% of the operators. Similar reduction rates are also observed in the other two open-source software.*

We apply Guideline 1, 3, and 4 to the configuration of the studied software, allowing an impact on fewer than 0%, 1%, and 5% of the existing operators, respectively. For Guideline 3, we convert a numeric parameter into an enumerative one if the parameter can be represented by no more than five options. Similarly, we convert an enumerative parameter into a Boolean if two options are sufficient to cover the settings in the field. We call the operators being "impacted" by the simplification if their current settings would be changed to slightly different settings. Note: It does not necessarily mean that the new settings would result in failures or performance degradation.

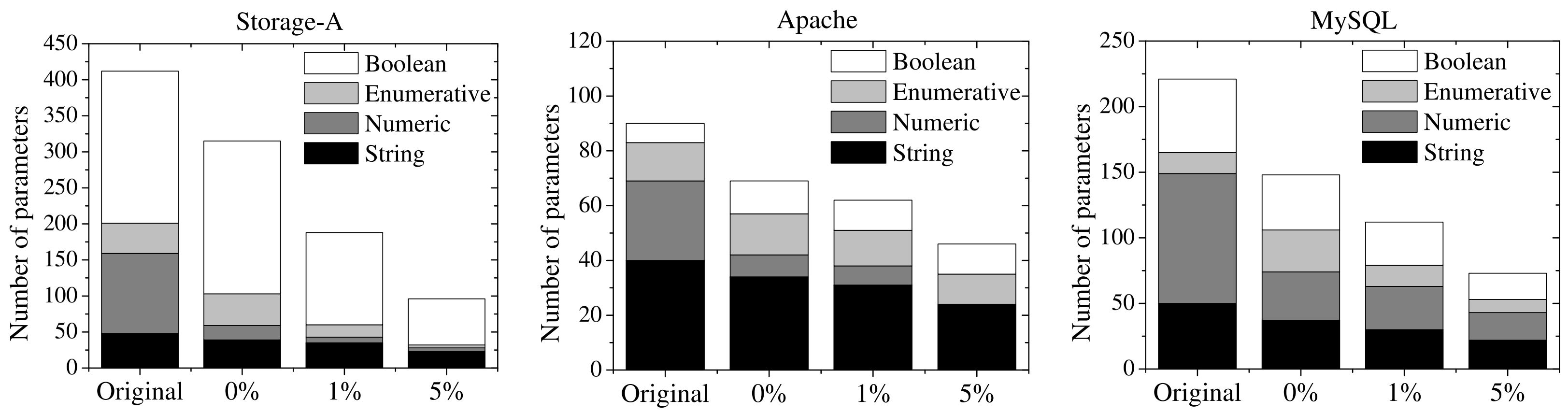


Figure 4.8. How much can we simplify configuration? The number of configuration parameters and their data types before and after we apply Guidelines 1, 3, and 4 in Table 4.10, with fewer than 0%, 1%, and 5% of the existing operators being impacted, respectively.

Figure 4.8 quantifies the effectiveness of the proposed configuration simplification methods. It shows the number of parameters and their data types after we apply the guidelines. As a first step in the direction of simplifying configuration, the results are promising, which also reflects the degree of the over-designed configuration.

4.6 Configuration Navigation

To deal with too many knobs, many software projects rely on the navigation feature to help operators find the right configuration parameters and their value settings. In this section, we conduct measurement study to understand the effectiveness of the navigation methods using real-world cases.

As discussed in §4.4.3, many operators encounter difficulties in finding or setting configuration parameters. As shown Table 4.11, the majority of these "difficulties" cases are about finding the configuration knobs rather than setting values. When an operator knows which knob to set, it is relatively easy to find the information from the manual pages or using the Unix `man` command), and to learn how to set it. Thus, configuration navigation should focus more on helping operators find the correct knobs.

4.6.1 Methodology

Navigation methods. We study three navigation methods—keyword search, Google search, and NLP-based navigation.

- **Search by keywords.** Search by keywords on top of manuals is a pervasive navigation practice. Many software projects provide build-in search utilities tied into documentation (e.g., the search box in MySQL online documents[10]). Even without specific support, operators can always rely on the search features offered by file readers/browsers to search keywords in PDF/HTML manuals.

[10]http://dev.mysql.com/doc/refman/5.6/en/index.html

Table 4.11. User-reported "difficulties" cases in finding configuration knobs versus setting the values. The closed "finding knobs" cases (e.g., the target knobs exist) are used for studying navigation methods in §4.6.

Software	Finding knobs	Setting knobs	Total
Storage-A	82.5% (47)	17.5% (10)	57
Apache	89.4% (42)	10.6% (5)	47
MySQL	84.8% (28)	15.2% (5)	33
Hadoop	82.9% (29)	17.1% (6)	35

- **Search on Internet.** Google search (or using other search engines such as Microsoft Bing) is another common practice to find the configuration knobs [19, 81]. Many software projects also provide search boxes that redirect users' queries to Google in their online manuals (e.g., Apache's online documentation[11]).

- **NLP-based navigation.** Recently, to help users find the right parameters, NLP-based navigation methods have been proposed [2, 78]. The idea is to build indexes for parameters based on their descriptions; users' queries are matched to indexed contents and the best matched parameters are recommended to the users.

Datasets. We select out "finding knobs" cases from the real-world cases studied in §4.4.3. These cases are the "finding knobs" ones shown in Table 4.11 We exclude cases of Storage-A because the case reports were written by the company's support engineers, and thus do not contain users' original questions/queries. In most of the "finding knobs" cases, the operators did find the target parameter(s) with the help of support engineers, or peer users from the online forums. We focus on these "closed" cases in our study and exclude the cases in which the target knobs do not exist. The number of the closed cases for Apache, MySQL, and Hadoop is 39, 25, and 26, respectively.

For each case, we use the original user-posted question as the original query. Every query is then filtered by common stop words which help remove meaningless

[11]http://httpd.apache.org/docs/2.4/

words, such as interrogative words, personal pronouns, articles, etc. Then, we convert each word in a query string to the root forms of the word based on WordNets [102]. The final query strings are used for studying all the three methods. For example, the original question, "*How do I configure the proxy to forward all requests*" is transformed into the query, "*proxy forward request*."

4.6.2 Effectiveness of Navigation

In this section, we measure the effectiveness of different navigation methods in dealing with the complexity of configuration. Note that these methods are not originally proposed for configurations in systems software.

Search by Keywords

Today's operators mainly rely on "greping" manual pages to navigate configuration parameters. Since no prior work has evaluated these two methods, we conduct measurement study to quantitatively understand their effectiveness as the starting point.

Finding 6(a): *Searching user manuals by keywords is not efficient to help operators identify the target parameter(s) to achieve the desired system behavior.*

As a user's query often contains multiple keywords (e.g., "*proxy forward request*"), she can first search for "*proxy*" and obtain all the pages containing it, and then search for "*forward*" and "*request*." Each keyword may be associated with multiple manual pages. We assume that an operator can find the target parameter(s) as long as she reads the page that contains the parameter (referred to as a *relevant page*). We study two search strategies: (1) *union* ($\cup$): returning all the pages contains at least one keyword, or (2) *intersection* ($\cap$): only returning the pages that contain all the keywords.

As shown in Table 4.12, the intersection approach is not effective. It returns relevant pages for only 15.4%$\sim$25.6% of the queries. The main reason is that the strategy

Table 4.12. The percentage (number) of queries for which the keyword search returns pages containing the target parameter(s).

Software	**% (#) of navigation cases**		**Total Cases**
	$\cap$\{returned pages\}	**$\cup$\{returned pages\}**	
Apache	25.6% (10)	79.5% (31)	39
MySQL	24.0% (6)	88.0% (22)	25
Hadoop	15.4% (4)	69.2% (18)	26

Table 4.13. The average number of returned pages per relevant page by keyword search

Software	**Avg. number of returned pages**	
	$\cap$\{returned pages\}	**$\cup$\{returned pages\}**
Apache	2	32
MySQL	15	102
Hadoop	9	139

is too *strict*. It *strictly* requires every keyword to appear on the manual pages of the parameters. On the contrary, the union approach returns relevant pages for the majority of cases; however, it also returns many irrelevant pages. As shown in Table 4.13, the average number of returned pages per relevant page can be as large as one hundred (in these cases, the user's query contains certain "common" keywords). It is impractical to read through these many pages to find the target parameter(s).

Some commercial management tools adopt the intersection method to provide navigation support on top of user manuals, e.g., Cloudera Manager (a commercial tool for Hadoop administration)[12]. As demonstrated above, they are too strict and thus limited in finding configuration parameters.

Google Search

We study Google search by sending the queries via Google search APIs. Then, we download the Web pages whose URLs are returned by Google. To make the searches explicit, we include the software name as a part of the query keywords. We analyze the

[12]http://www.cloudera.com/content/cloudera/en/products-and-services/cloudera-enterprise/cloudera-manager.html

Table 4.14. Effectiveness of Google search. The percentage (number) of queries for which Google returns useful Web pages in top-five search results. "At the time" excludes pages posted after the original cases to emulate the situation when operators encountered navigation issues.

Software	**% (#) of queries w/ useful Web pages**		**Total**
	At the time	**Postmortem**	
Apache	35.9% (14)	74.4% (29)	39
MySQL	40.0% (10)	80.0% (20)	25
Hadoop	26.9% (7)	46.1% (12)	26

top-5 Web pages returned by Google and examine if they are useful (existing studies show that the top-5 results attract most clicks [135, 136]). A *useful Web page* must meet the following two criteria: (1) containing the target parameter(s) (the *recall* metric); and (2) containing no more than five other parameters (the *precision* metric).

Finding 6(b): *Google search can provide useful information for 46.1%$\sim$80.0% of the historical configuration navigation issues. However, it is less efficient in navigation parameters of less popular software or new issues. The majority of resources on the Web that host useful information for navigation are the contents contributed by operators, such as Q&A forums and blog articles.*

Table 4.14 shows the effectiveness of Google search for configuration navigation. Google returns useful pages in the top-5 results for 46.1%$\sim$80.0% of the queries. In other words, if these *historical* navigation issues are encountered by the new operators, 46.1%$\sim$80.0% of them could be resolved by Google search. Hadoop has a remarkably low number (more than half of the historical cases cannot resolved by Google), mainly for two reasons. First, Hadoop has a much smaller user base with less online resources, compared with Apache and MySQL. Second, the primary Q&A sites of Hadoop is its official user mailing list. However, mailing list archives have very low page ranks, making them less "visible" by Google search.

Table 4.15. Breakdowns of the sources that host the useful Web pages.

Source	Example	Percentage
Q&A forums	ServerFault.com	37.4%
Blogs & articles	Articles on Blogger.com	22.8%
Official docs	MySQL online docs	22.8%
Third-party docs	Hortonwork's Hadoop docs	8.1%
Docs of other SW	PHP's docs on MySQL conn.	5.7%
Others	Wiki, Bugzilla	3.2%
Mailing lists	Hadoop's mailing-list archive	0.0%

Also, we emulate the situation when the operator encountered a navigation issue and searched Google, by excluding Web pages posted after the original user question. As shown in Table 4.14, no more than 40% of these issues can be resolved by Google. Many of the returned Web pages "*at the time*" are online manuals and tutorials that include too many configuration information, which has similar efficiency as searching keywords on top of manuals (c.f., §4.6.2).

To understand what types of Web pages are useful for configuration navigation, we classify the useful Web pages returned by Google based on their types, as shown in Table 4.15. Remarkably, user-generated pages contribute to more than 50% of these useful pages, such as Q&A posts, blogs articles. These Web pages usually record the operators' experience and solutions to specific configuration problems, and only contains a small set of relevant parameters, which is more useful for navigation (compared with online manual pages that list parameters one by one). Therefore, methods to leverage user-generated contents, especially those with low page ranks (e.g., mailing-list archives) is desired for configuration navigation.

NLP-based Navigation

To help operators get the right configuration knob (preference), NLP-based navigation methods have been proposed (e.g., PrefFinder [79] and COX [2]). These NLP-based navigation methods take a user's query in natural languages as the input, and return the configuration parameters relevant to the query. This is achieved by parsing, analyzing, and indexing the information of every parameter (e.g., from its manual entries) using NLP techniques, such as stemming, stop-word filtering, text normalization, synonym expansion, etc. To return the relevant parameters, the navigation engine ranks the parameters by *scoring* how well they match the query.

We study the efficacy of NLP-based navigation based on COX (PrefFinder is not open sourced). COX is a configuration navigation library on top of Apache Lucene [4]. It provides utilities to extract the texts for every configuration parameter by breaking manual pages, and allows us to change the parsing, analyzing, and scoring methods.

In addition to indexing and matching, we also take the field configuration statistics into account. The original match score is *boosted* by the *popularity* of the parameter, defined as the percentage of operators who set the parameter among all the operators (same as in §4.4.1). The idea is to boost popular parameters with higher ranks if they match users' queries. Also, we assign a lower *scale* 0.4 to contents from manual entries while parameter names have scale 1.0.

Finding 6(c): *Well-engineered NLP-based navigation can return the target configuration parameter for more than 60% of the historical navigation issues. Boosting the results with the statistics of configuration settings in the field can significantly improve the performance of NLP-based navigation.*

Figure 4.9 shows the results of the NLP-based navigation using different combination of information sources (parameter names, statistics of configuration settings in the field, and manual pages). We observe that the navigation only based on parameter names has poor performance (much worse than the dataset in [79][13]). The reason is that in our dataset of systems software, many of the queries do not contains any keyword

[13]http://cse.unl.edu/~myra/artifacts/PrefFinder_2014/

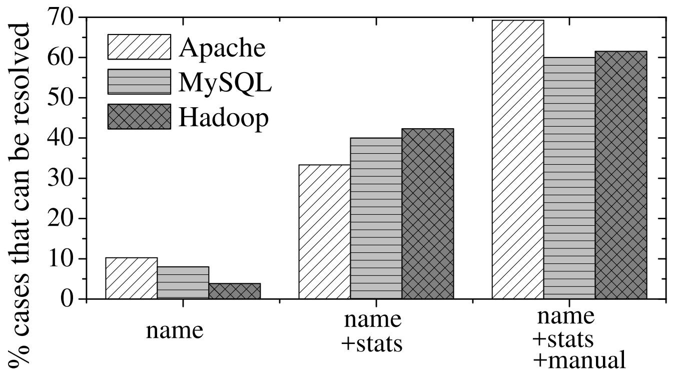


Figure 4.9. The performance of NLP-based navigation using different information sources. We consider a case can be resolved if the target parameter can be returned in the top-5 navigation results.

appearing in the parameter name, which is different from queries to desktop software configuration [79]. In desktop software, users are usually able to specify useful keywords like "*color*," "*tab*," "*cache*" (which are parts of the target parameter's name), while the queries here are higher level intentions such as "*speedup insert performance*" (target parameter: `max_heap_table_size` and there are more than 80 size-related configuration parameters in total). In this case, applying the statistics of field configurations brings significant performance improvement, because *a configuration parameter that is used by many operators are likely to be needed by the current operator.* In addition, the results show that leveraging contents from manual entries are useful—manual entries bring additional information about the parameter.

Overall, our NLP-based navigation implementation resolves more than 60% of the real-world cases. We manually examine the unresolved queries and find most of them indeed miss the keywords in the contents of the target parameter. First, some queries are vague or misleading (even for human experts). For example, `alias`-related parameters are returned for the query, "*alias url without use host*," but the target one is related to virtual hosts. Second, some queries require domain-specific knowledge beyond the information base. For example, it fails to associate SSL with "encrypt," and fails to return SSL-related parameters for "*encrypt network channel*." This limitation can potentially be addressed by using word-cluster based techniques to capture "concepts" instead of "keywords."

4.7 Discussion

4.7.1 Implications and Incentives

Although motivated by operators' configuration problems, reducing configuration space and simplifying configuration not only help operators' configuration difficulties and problems, but also would bring tremendous benefits for software vendors by relieving their burden of testing, error detection and troubleshooting.

Testing software with large configuration space is extremely challenging. The number of possible configuration settings is an exponential function of the number of parameters and their value space, which makes it infeasible to test exhaustively. This is known as *configuration space explosion* [99, 184]. To address this problem, a series of pioneer works have been proposed, including pruning the configuration space [75, 128, 141], selecting typical configuration values [46, 121, 184], prioritizing certain important configurations [121, 145], and reducing the number of test cases [120].

As shown in our study, many configuration parameters are not in real use, i.e., a large portion of the existing configuration space is not touched by operators. Removing the unused configuration can significantly relieve the burden of software testing in the context of configuration space explosion, as complementary to the existing testing methods. Prioritization becomes natural, considering the actual configuration usage statistics in the field—the parameters/values set by more operators should have higher popularity than the ones set by few operators.

The smaller configuration space also benefits operators and support engineers for misconfiguration detection and troubleshooting. Small configuration space comes with small error space, which not only makes it easy for operators to examine and find the errors, but also make the automatic detection and troubleshooting procedures more efficient. Also, with smaller error space, the detection and troubleshooting tools can be more focused and targeted. Most importantly, with simplified configuration, operators are likely to have less configuration difficulties and problems, which in turn results in lower support cost for software vendors and developers.

With these incentives, we advocate software vendors to take actions in simplifying existing configuration and providing new configurations more cautiously with user-centric design philosophy.

4.7.2 Further Simplification

This chapter mainly investigates the feasibility and opportunity of simplifying configuration in the aspect of reducing the configuration space (including both the parameter space and the value space). However, it is important to note that the configuration space is not equivalent to the entire configuration complexity. In other word, the efforts to simplifying configuration should not be limited to reducing the configuration space (which is only our first step towards addressing this problem).

Besides the large configuration space, other known root causes of configuration complexity include ambiguity and inconsistency of configuration semantics [56, 178], dependencies among multiple parameters and software components [39, 89, 126, 132, 173, 190], and poor system guidance and feedback [74, 177, 179]. It remains as our future work to understand and address these aspects of configuration complexity perceived by operators, with the goal of making configuration simple, efficient, and less prone to errors. Similar as the study in this chapter, we believe the key towards addressing these problems is to follow the user-centric philosophy—to understand operators' difficulties and problems in the field and to design configuration from the operators' perspectives. After all, configuration is one type of user interface that are used by operators to control and customize the system behavior.

4.7.3 Intent-based Configuration

One promising direction to simplify configuration is intent-based configuration design where operators only express high-level configuration intent which will be translated into concrete values of low-level configuration knobs. Intent-based configuration design can free operators from setting thousands of tedious knobs and thus focus on declaring the intended system behavior. In fact, prior work has shown that intent-based configuration can be achieved for network configurations where network-wide routing policies can be automatically translated into device-level configurations [25, 110, 151].

However, it is not clear how to build intent-based configuration for cloud and datacenter systems. The main challenge towards a generic solution is that unlike network configurations (e.g., those specifying routing paths) that have well-defined semantics and have direct links from the value settings to the system behavior (forwarding/dropping packets), the configurations of software are much more diverse and the impact of configuration values (in terms of system behavior such as performance, reliability) is more subtle. It remains our future work to explore intent-based configuration for common types of configurations in cloud and datacenter systems.

4.8 Summary

The configuration of system software has become increasingly complex. To advocate cautious and disciplined thinking in configuration design, this chapter has provided quantitative evidence for the over-delivered (or under-exploited) flexibility represented by configuration parameters. By studying the large-scale configuration settings of real operators, we have revealed a number of findings, leading to a few guidelines for simplifying configuration. We also studied configuration navigation as an intermediate solution, if the simplification process takes time.

We hope that our work can inspire developers to design system configuration with the user-centric design philosophy, and carefully balance simplicity (usability) and flexibility (configurability). Similar to UI/UX design, it is important for developers to collect feedback from operators and think from their perspectives, before introducing yet another knob. Feedback loops should be initiated and followed to help developers improve the usability of their software systems. The original paper [175] has been widely quoted within the company of Storage-A and used to make decision on improving configuration design for reducing their customer's misconfigurations.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2015. Xu, Tianyin; Jin, Long; Fan, Xuepeng; Zhou, Yuanyuan; Pasupathy, Shankar; Talwadker, Rukma. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Related Work

“To learn without thinking is blindness; to think without learning is idleness.”
—Confucius

This dissertation research stands on the shoulder of prior work in a number of ways. This chapter describes prior work, how it has inspired this dissertation, and the contributions that this dissertation offers beyond existing research. §5.1 discusses auto-configuration that aims at fundamentally ruling out potential mistakes. §5.2 discusses approaches and practices that check correctness of configurations in the field. §5.3 discusses how to deal with misconfiguration-induced failures and anomalies.

As stated in Chapter 1, this dissertation fundamentally differs from the prior work discussed in this chapter in terms of its perspectives—it focuses on hardening the internal defense of software systems through better design and implementation, while the prior work mostly targets on building external tools and practices.

5.1 Automating Configuration

One ambitious solution to misconfigurations is to free operators from configuration by eliminating the need of manual configuration efforts, which prevents human mistakes in the first place. On the other hand, completely automating configurations is not trivial. For example, it is hard for a system to automatically determine the configurations that requires information outside the system itself (e.g., connection configurations of backup services); it is also controversial whether systems should automate critical configuration settings that may permanently change system states or data [164]. We discuss two main approaches that aim at automating certain types of configurations.

5.1.1 Tuning Performance Configurations

One specific class of hard-to-set configuration parameters are those related to system performance. Large systems usually include a large number of performance-related parameters (e.g., memory allocation, I/O optimization, parallelism levels, etc). Their impact on performance is often poorly understood due to inexplicit correlations and interactions both inside the system and across multiple system components and runtime environments. Consequently, tuning performance configurations is challenging, especially for operators with limited knowledge of system internals. Also, with the explosive configuration space, it is prohibitively difficult and costly to test out every value combination and then select the best ones. Mature systems provide *default* values for these configuration parameters. The default values are usually carefully selected by the developers based on their experience and/or in-house performance testing results. Ideally, good default values can satisfy common workloads and system/hardware settings. However, given dynamic workload and system runtime, it is hard for the static default values to deliver *optimized* system performance [97, 161].

There is a wealth of literature on performance tuning by selecting configuration values. The basic idea is to model the performance as a function of configuration settings, as the following equation[1],

$$p = F_W(\vec{v}), \text{ where } \vec{v} = (v_1, v_2, ..., v_n) \tag{5.1}$$

[1]This model can be extended to include other factors. For example, [67] models the performance of one MapReduce job W as $p = F_W(\vec{v}, \vec{r}, \vec{d})$ where $\vec{r}$ is the allocated resources and $\vec{d}$ represents other statistical properties of data processed by W.

Here, F_W is the performance function of a workload type W; $\vec{v}$ is the configuration value set consisting of the value of each i-th configuration parameter (the number of parameters is denoted as n); p is the performance metric of interest (e.g., execution time, response latency, throughput). With such a model, the performance tuning problem is to find the value set $\vec{v}^{\,*}$ that achieves the best performance (i.e., *argmax*),

$$\vec{v}^{\,*} = argmax\, F_W(\vec{v}) \tag{5.2}$$

Since the performance function F_W is often unknown or does not have a closed form, a number of *black-box* optimization algorithms have been proposed, including a variety of sampling and searching algorithms [114, 171, 183, 198], gridding [67], Bayesian optimization [8], and machine learning [7, 109, 139]. Other studies try to capture the performance characteristics using specific models. For example, Duan et al. use Gaussian process to represent the response surface of F_W in relational database systems [45]; Zheng et al. apply dependency graphs to describe the performance impact of different parameters for web applications [194].

These studies provide theoretical guidance for modeling performance impact of configuration settings. However, it is fundamentally difficult to precisely model the system performance in the field due to many unexpected and confounding factors [161]. Consequently, some of the models are limited to certain workloads and environments, making them less appealing in practice.

5.1.2 Reusing Configurations

As configuration is difficult, the experience and efforts towards the correct configuration solutions should be shared and reused. Often, a configuration task that is difficult for an operator has been encountered and solved by others before. Most importantly, the previous working solutions are likely to be helpful (and even directly applicable) to the new systems under configuration.

Configuration file templates. The basic form of reusing configurations is via *templates*. A typical configuration file template is a working solution for one particular use case. Templates can also be made for different hardware or environment conditions. For example, MYSQL once provided five templates for servers with different memory sizes, ranging from less than 64 MB to larger than 4 GB. These templates save operators' efforts of tuning memory-related configuration parameters which is difficult but critical to performance. Templates not only can be provided by system vendors, but also can be distributed by third-party tool providers or among user communities. For example, the user communities of configuration management tools (e.g., PUPPET and CHEF) have the tradition of sharing configurations for a variety of common software systems.

Record-and-replay systems. Static configuration file templates have two major limitations. First, they cannot capture configuration *actions*, for example, setting file permissions, installing software packages, creating new users and groups. These actions can be a necessary part of the configuration solution. Second, the solution provided by the templates may not work for the new system; applying a wrong solution may have side effects—changes of the system states. In this case, the changes need to be undone. Manually undoing system changes is not only tedious but also difficult as operators may not be aware of some implicit changes of system states.

Record-and-replay systems are proposed to address the above two limitations and to make configuration reuse fully automated. The basic idea is to *record* the traces of a working configuration solution on one system, and then *replay* the traces on other systems under configuration. If the replay fails (e.g., not passing predefined test cases), the system will be automatically rolled back to its original state. Since the traces for the same configuration problem may differ (e.g., due to system and environment settings), multiple raw traces could be merged to construct the canonical solution.

AUTOBASH [149] and KARDO [86] are two representative record-and-replay systems designed for different use cases. AUTOBASH is designed for recording and replaying configuration tasks performed in Bash (or other UNIX-style shells). It leverages kernel speculation to automatically try out solutions from the solution database one by one until it finds a working one. The kernel speculator ensures that the speculative state is never externalized, i.e., it isolates AUTOBASH's replay activities from other non-configuration tasks. AUTOBASH requires operators to provide test cases and oracles (called predicates) in order to decide whether or not the configured system is correct. Later, Su et al. [150] further propose automatic generation of predicates by observing the actions of troubleshooting processes. KARDO is designed for automating GUI-based configurations on personal computers. It records the window events when configuration tasks are performed based on OS-level accessibility support. KARDO analyzes the raw traces and identifies the window events specific to the task and the events for state transition events. It then constructs the canonical solution for a configuration task; the canonical solution works for systems with different internal states.

5.1.3 Discussion

In addition to automating configuration, Chapter 4 studies a more aggressive approach—simplifying configurations based on how they are used in the field. Essentially, configurations are introduced for the purpose of providing flexibility, with the tradeoff as increasing complexity. If the flexibility is not needed or appreciated, the complexity is not worthwhile. Unfortunately, Chapter 4 shows that many of today's configuration knobs are neither necessary nor worthwhile. Such configurations should be eliminated. Chapter 4 also discusses other approaches for simplifying configurations (e.g., intent-based configurations), which shares the same goal as auto-configuration.

5.2 Checking Correctness

The main approaches that check the correctness of configuration settings in the field include *static* misconfiguration detection and *dynamic* online testing.

5.2.1 Detecting Misconfigurations

As discussed in Chapter 3 (§3.2.2), today's misconfiguration detection methods are mostly *rule based*—the detector checks the configuration values against a set of predefined correctness *rules* (also known as *constraints*). If a configuration value does not satisfy these rules, it will be flagged as a misconfiguration. The key challenge of rule-based detection is to define and manage useful rules.

Defining rules. Most mature systems implement built-in configuration checking logic to detect syntax and format errors against basic rules. As observed by [108], the checking is useful in detecting operators' configuration mistakes. Certain domain-specific rules have been proposed in different system domains. For example, RANGE-FIX [173] model the data-range rules among multiple parameters, which helps detect and fix invalid settings. Feamster et al. [50] define two general correctness rules for BGP (Border Gateway Protocol) configurations: the path visibility and the route validity. Based on the two rules, their tool RCC detects BGP misconfigurations by statically analyzing BGP configurations. CONFVALLEY [71] provides a simple, domain-specific language called CPL (Configuration Predicate Language) for operators to write configuration checking logic in a compact, declarative fashion. Compared with traditional imperative languages, CPL can significantly reduce the efforts of writing checking code.

Generally, manually specified rules face the following two problems. First, it is hard to make predefined rules complete [84]. In the study of [108], though Apache's checker detected a number of misconfigurations based on predefined rules, it did not check a misconfigured file path, causing `mod_jk` to crash. Second, it is costly to keep the rules updated with the software evolution, since it requires repeating the manual efforts. As shown in [192], configuration rules could be obsolete with code changes.

Learning rules. To address the problems derived from the manual efforts of defining rules, a number of research proposals provide automatic approaches to learning configuration rules. The basic idea is to learn the "common patterns" from large volumes of configuration settings collected from a large number of healthy system instances. The patterns shared by most of the healthy instances are assumed to be correct, and will be used as the correctness rules for misconfiguration detection. A configuration setting that does not follow these patterns is likely to be misconfigurations.

With this idea, a variety of machine learning techniques are applied to learn configuration rules from different types of configuration data. CODE [189] learns access patterns of Windows Registry configurations which indicate the appropriate configuration events that should follow each context (a series of events). Based on these patterns, CODE can sift through a sequence of configuration events and detect the deviant ones as misconfigurations. Palatin et al. [115] propose a distributed outlier detection algorithm to detect misconfigured machines (i.e., the outlier machine) in grid systems based on log messages, resource utilizations, and configuration settings. Kiciman et al. [85] apply unsupervised clustering algorithms on Windows Registry configurations in order to learn configuration constraints based on the Registry-specific structures and semantics.

One concern of automatic learning is the accuracy. Inaccuracy results in false rules which cause false positives in detection. False positives do matter. Bessy et al. [28] report that in reality, operators ignore the tools if the false-positive rate is more than 30%. Note that misconfiguration detection is performed before system failures and anomalies. At the time, operators tend to be optimistic. This psychological issue fundamentally makes misconfiguration detection different from the postmortem failure diagnosis (when operators or support engineers are desperate and willing to look at any hints). Unfortunately, learning-based approaches often cannot provide sufficient accuracy and are difficult to tune. Many of the learning-based approaches discussed above also require manual intervention. To address the inaccuracy of learning, EnCore adopts a template-based learning approach [190]. In EnCore, learning is guided by a set of predefined rule templates which enforce learning to focus on patterns of interests. In this way, EnCore filters out irrelevant information and reduces the false positives.

As discussed in Chapter 3 (§3.2.2), the more fundamental limitations of machine-learning based approaches derived from their requirement of a large collection of independent configuration settings deployed at hundreds of machines. Such deployment works where a piece of software is distributed and install on the customers' machines. However, in the modern cloud era where cloud and datacenter systems providing services (rather than software itself), configurations are typically propagated from one node to all the other nodes. Thus, the settings from these nodes are not independent, and thus not useful for learning. Furthermore, learning-based approaches do not work well with configurations that are inherently different from one system to another (e.g., environment configurations such as domain names, file paths, and IP addresses) or incorrect settings that fall in normal ranges. In general, they also cannot differentiate customized settings from erroneous ones.

In addition, we have to confront the fact that misconfigurations are still hard be ruled out. Besides the limitations of the existing approaches, certain misconfigurations can hardly be detected even by static rules without understanding the runtime information or global policies. Such misconfigurations are referred to as *legal* misconfigurations [185], which are not rare. Among the 434 real-world parameter-misconfiguration cases studied in [185], "*a large portion (46.3%∼61.9%) of the parameter misconfig-*

urations have perfectly legal parameters but do not deliver the functionality intended by operators. These cases are more difficult to detect by automatic checkers and may require more training or better configuration design." Misconfigurations related to resource allocation, performance, access control and security are such examples.

5.2.2 Online Testing

Online testing is complementary to static misconfiguration detection discussed in §5.2.1. The basic idea is to observe the effect of the configuration settings by testing them out in a separate testing environment or one part of the production system [155], before rolling out the configuration settings and making them visible to the entire production systems. Online testing is especially useful when the effect of configuration settings is not well understood or hard to model. It allows operators to apply the trial-and-error methods on different settings. Moreover, performing online testing, as a rehearsal, is necessary for configurations that are mission critical or cannot be rolled back, for example, formatting disks that will remove customers' data.

The main challenge of online testing is to create an environment that has the same (or similar) characteristics of the production systems including the workloads, the execution environments, and the underlying infrastructure. If the testing environment cannot capture these characteristics, the testing results are less trustworthy. However, such construction is very challenging because of resource constraints and the separation from the production environments, especially for large-scale systems. Welsh once shared his experience at Google [169], "*Of course we have extensive testing infrastructure, but the 'hard' problems always come up when running in a real production environment, with real traffic and real resource constraints. Even integration tests and canarying are a joke compared to how complex production-scale systems are.*" As summarized in [30], online testing needs to be *comprehensive* for determining whether or not the configuration settings are valid, *isolated* to the production environments, and *efficient* in terms of resource usage.

Nagaraja et al. [108] propose to integrate such testingi (they used the word "validation") as an extension of the production systems, which addresses the separation between the validation and the production environments. In their prototype designed for web services, the system is divided into two logical slices: the *online slice* that hosts the production services and the *validation slice* which validates new configuration settings. The two slices are connected through a proxy that duplicates user requests from the production services to the validation components. In this way, the validation components can test new configurations under real workloads. The same idea is applied in [112] for database systems.

However, setting a separate testing environment into production could be too costly and thus not affordable for systems at scale, especially today's cloud and datacenter systems. Therefore, techniques that apply the tests directly on the production systems are proposed and adopted in recent years. KRAKEN [161] is Facebook's load testing framework that continuously shifting live traffic to test systems ranging in size from individual servers to entire data centers. Such load tests can effectively identify regressions, address load imbalance and resource exhaustion across Facebook's infrastructure (many of these issues are caused by misconfigurations [161]). CHAOS MONKEY and the Simian Army [21, 159], MOLLY [9, 10], and other chaos engineering techniques [90, 130] are used as failure drills in which faults are deliberately injected in the production system to expose the system's vulnerabilities related to failure resilience. Note that load testing and chaos engineering are not specific to misconfigurations. The goal is to expose the problem regardless of the root causes of the problems.

5.2.3 Discussion

PCHECK in Chapter 3 is greatly influenced by both misconfiguration detection and online testing. On one hand, we prefer to have a *light-weighted* detection approach that can be run cheaply and regularly. On the other hand, we want to keep the advantage of online testing that can validate the configuration values on the native system environment and find problems that cannot be covered by traditional machine learning based approaches. Since PCHECK is designed specific for configurations, we do not need to execute the entire system programs but only the program "slices" that will use the configuration values in later execution.

5.3 Dealing with Misconfiguration-Induced Failures

When misconfigurations escape from all the defenses and cause failures, the operators or engineers have to deal with the failures by (1) troubleshooting the misconfigurations based on the failure symptoms; and (2) recovering the failed systems. Theoretically, recovery should be done after the root causes are discerned. However, in practice, as the diagnosis can take hours [54], recovery and fallback is typically done in parallel in order to minimize the downtime short. One common approach is to recover to a previous configurations (and corresponding code) that is known to be good [97].

5.3.1 Troubleshooting Misconfigurations

Today's misconfiguration troubleshooting still mostly relies on manual efforts involving examining configuration settings, analyzing log messages, checking system states, searching knowledge base articles, etc. If operators cannot successfully diagnose the problem by themselves, they have to call support engineers for assistance. This often falls into more costly diagnosis cycles, including collecting site information and remote debugging. Therefore, it is highly desired to have automatic solutions and tool support to facilitate the troubleshooting process based on the failure-site information (e.g., log messages, stack traces, coredumps, etc).

Troubleshooting misconfigurations has to address the following two challenges. First, since operators may not have debugging information (e.g., source code) or the expertise of debugging complex systems, troubleshooting cannot follow the similar interactive process as software debugging, which asks developers to examine the program runtime execution and reason out the root causes (e.g., using GDB). Second, the goal of misconfiguration troubleshooting is to pinpoint the erroneous configuration settings rather than the bugs inside the software. Thus, existing diagnosis support (e.g., reconstructing the execution paths and reproducing the failures) may not be sufficient or useful for helping operators find the misconfigurations. Troubleshooting misconfigurations needs to go extra steps from errors in the program to errors in the configurations.

Causality analysis. Several troubleshooting approaches attempt to automatically identify the erroneous setting as root causes of the failures, analyzing the causal relationships between the settings and the failures. From the system's perspective, configuration settings are one type of system inputs that are read and stored in the system variables, and then used during the runtime execution. As they deviate from the normal executions, misconfigurations lead to the *failure executions*, i.e., executions ended with system failures. Causality analysis attempts to reason out the causality between the configuration settings (cause) and the failure execution (effect).

CONFAID [18] is a representative misconfiguration troubleshooting tool based on causality analysis. To troubleshoot a misconfiguration, CONFAID first instruments the program binaries to insert the monitor code which will record the information flow during the program execution. Then, it re-runs the program in order to reproduce the failures using the instrumented binaries. Based on the recorded execution information, CONFAID identifies the causal dependencies between each configuration setting and the failure. The configuration setting with stronger dependency to the failure is more likely to be the root cause of the failure. On top of CONFAID, X-RAY [16] is built for troubleshooting performance related misconfiguraitons. X-ray records the performance summarization of each configuration setting during the program execution, quantifying the performance impact caused by the setting. The settings with the bigger impact are considered to have stronger causality with the performance anomaly.

Different from CONFAID, CONFANALYZER [123] and CONFDEBUGGER [44] are static approaches that precompute the causal dependencies between configuration parameters and potential failure points. Thus, it does not require instrumenting the binaries and reproducing the failures. These approaches analyze the data- and control-flow of each configuration parameter on each potential execution paths in the program. It stores the mapping from the potential failure paths to the configuration parameters in a table. To diagnose a failure, they first reconstruct the failure execution path based on log messages and stack traces; then it queries the table using the failure path to obtain the dependent configuration parameters whose settings are suspected to be erroneous.

Signature-based approaches. In the early 2000, Microsoft investigated a series of signature-based troubleshooting approaches, striving to automatically reason out the misconfigurations by comparing the signature of failure executions with the reference signatures.[2] Reference signatures are usually recorded from executions with known system behavior (mainly WINDOWS): they can be either normal executions or the known failure executions recorded from previous troubleshooting efforts.

PEERPRESSURE [165] and its predecessor STRIDER [168] compare the failure

[2]A *signature* refers to the information recorded at the system runtime. Its semantics depend on the signature collector of the troubleshooting approach. For example, the signature could be the configuration settings [168], system call traces [186], and dependency set [17].

signature with signatures of normal executions. Upon a failure, a client-side TRACKER captures the configurations as the signature of the failure execution and sends it to PEER-PRESSURE. These configurations are treated as misconfiguration suspects. Next, PEER-PRESSURE queries a centralized signature database and fetches the respective settings of the configurations that were collected from the sample machines. Then, it uses Bayesian estimation to calculate the probability of an configuration value to be erroneous and outputs the ranks of every configuration.

Differently, Yuan et al. [186] propose to compare the signature of the failure execution with other known failure signatures. If a match is found, the previous troubleshooting efforts can be reused to address the current misconfigurations. In their proposed tool, a TRACKER collects system call traces as the signature at the client side. SVM (Support Vector Machine) is used to match the signature with the category generated from the signatures of known problems. The solutions of known problems will then be returned to operators. The idea of signature-based analysis has also been applied in a number of other tools, such as NETPRINTS [6] and CONFDIAGNOSER [191].

Checkpoint-based approaches. Checkpointing can be used to record how a system transitioned from the working state to the failure state when a failure occurs. To troubleshoot the configuration error that causes the failure, the operator can compare the state of the system immediately before and after the failure using the UNIX `diff` command [170]. CHRONUS [170] automates the search for the time when the system transitions from a working state to a failure state. It only checkpoints the persistent storage based on a time-travel disk that has relatively low overhead compared with whole-system checkpointing. CHRONUS uses a virtual machine monitor to instantiate, boot, and test historical snapshots of the system in order to determine if a system snapshot represents a working state. SNITCH [101] leverages FDR [163] to construct the timeline views of the system states. The timelines are useful for validating the root cause hypothesis that the decision tree-based troubleshooting procedures suggest.

5.3.2 Failure Recovery

Recovering from failures is non-trivial. Common recovery techniques such as rebooting based ones (e.g., MICROREBOOT [35]) that reclaims system resources and clearing runtime states, is less helpful because misconfigurations typically reside in persistent configuration files or databases. Rebooting cannot get rid of them. Rolling the system back to a recorded healthy state (snapshot) could be a potential solution based on techniques such as application checkpointing, backup/restore, and system-wide UNDO [33]; however, the runtime overhead and storage overhead could be expensive.

In industry, there are two major practices of recovering failures induced by misconfigurations. First, if the root cause is likely to be caused by configurations (e.g., the failure occurs after a configuration change), the recovery reverts the change and rollback to an old configuration that is known to be good [54, 97]. The hard part is the dependencies—the configuration change could (and often) have dependencies with changes of code and system states. Simply reverting the configuration values does not work. Second, to avoid degraded user experience during the recovery process, operators traffic is often redirected from the failing components to other healthy components in levels of nodes, clusters. data centers, and regions [29,54]. This requires diversity in the entire systems and carefully designed configuration/code rollout process [29, 155].

5.3.3 Discussion

Dealing with failures is hard, especially those with misconfigurations. Such bitterness is the motivation that drive the theme of this dissertation. Fundamentally, we hope to avoid paying the cost for dealing with failures caused by configuration errors, but to build more reliable systems that can anticipate and defend against misconfigurations, and to prevent misconfigurations in the first place by better design methods. As we have demonstrated in the previous chapters, even for legacy systems, this can be achieved systematically and automatically via exposing the vulnerabilities (Chapter 2), enabling early detection (Chapter 3), and simplifying configuration space (Chapter 4).

5.4 Summary

This chapter discusses prior work closely related to this dissertation, including eliminating the need of configurations, checking correctness of configuration settings, and dealing with misconfiguration-induced failures. Note that there are many other topics and work that are also critically important to configurations. Examples include configuration management and deployment [14, 15, 42, 47, 137, 138, 155] and configuration languages/APIs [53, 95, 122]. More are discussed in the original survey article [180].

Chapter 5, in part, is a reprint of the material as it appears in ACM Computing Surveys, Vol. 47, No. 4, Article 70, 2015. Xu, Tianyin; Zhou, Yuanyuan. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conclusion and Future Work

This dissertation presents the principles of systems design and implementation for building reliable cloud and datacenter systems in the face of misconfigurations. We demonstrated that enabling the principled systems design and implementation can (1) proactively detect real-world, high-impact configuration errors; (2) effectively assist operators to fix misconfigurations in the field; and (3) significantly reduce the complexity and error-proneness through configuration design. These principles are grounded in our experience building the tool support that that can automatically and systematically apply these principles to existing cloud and datacenter systems.

Specifically, this dissertation has shown that understanding how configuration values are used inside the software systems opens tremendous opportunities for building the hardening techniques against misconfigurations. SPEX demonstrates that the constraints of configuration values required by a system can be inferred automatically through static code analysis, and can be leveraged to expose misconfiguration vulnerabilities through misocnfiguration-inject testing. This enables developers to fix the vulnerabilities and enable systems to react gracefully to misconfigurations in the field. PCHECK shows that it is feasible to emulate late execution that will use configuration values by extracting the corresponding instructions, and more importantly to detect latent configuration errors early based on their manifestations exposed in the emulated execution.

This dissertation also shows that by treating configuration as a user interface and applying user-centric design philosophy, one can significantly improve the usability and reduce error-proneness of software configuration. We present configuration simplification, as well as navigation techniques and tool support (COX), based on the characteristics of how configurations are actually used in the field.

Still, the contributions of this dissertation present the first step towards hardening software systems' own defense against misconfigurations. More work needs to be done to build complete, solid defense in depth. First, SPEX and PCHECK only focuses on misconfigurations that compromise the correctness of software programs. For example, PCHECK reports configuration errors based on anomalies manifested during emulated execution. On the other hand, as reported in prior work [155, 169, 185], a larger percentage of real-world misconfigurations are *legal*. Legal misconfigurations have correct syntax and semantic, but do not deliver intended system behavior (e.g., security guarantee, resource constraints, or performance goals). Unfortunately, there is few understanding in proactively defending software systems against them. The key to tackling legal misconfigurations is to (1) specify the intended system behavior and check configurations against it, and (2) understand potential impact of configuration changes. The open question is that how to achieve these systematically and automatically and incorporate application-specific information to make the results useful.

Moreover, the current defending techniques (including SPEX and PCHECK) stop when the misconfigurations are discerned. After a configuration error is detected or manifested, operators need to fix the errors. The efficiency and correctness of fixing misconfigurations significantly impact the time to recovery. An open question is the feasibility of automatically correcting misconfigurations by altering the values of configuration parameters in configuration files. This is non-trivial given the enormous space of potential values. To go one step further, it would be useful to reload the modified configuration values into the system and re-execute the related code, in order to recover the abnormal execution, in a similar way as how re-execution based approaches (e.g., [119, 129]) recover other types of system errors like memory errors.

In fact, the fundamental challenges in addressing misconfigurations are less about finding a decent value in the configuration space, but more about understanding complex, dynamic systems at scale. As cloud and datacenters systems typically consist of large numbers of heterogeneous components across stacks and nodes, the impact of configurations often manifests through indirect component interactions and thus cannot be determined by per-component testing or even small-scale canaries [91]. Moreover, configuration errors may not be manifested through independent anomalies, but being correlated with other errors such as bugs and even gray failures [72]. Understanding the interactions and dependencies between multiple software components are the key to foreseeing the impact of configuration changes, which remains a grand challenge.

Appendix A

Operation Disciplines and Implications

This appendix explains different operation disciplines including *traditional system administration*, *DevOps*, and *site reliability engineering*. It also discusses the implications to this dissertation.

Historically, configurations are managed by system administrator (sysadmins) who are responsible for managing software systems. Unlike software developers, sysadmins are known to have a diverse background due to the lack of education programs [31]. According to Wikipedia.org, "there is no single path to becoming a sysadmin [...] a sysadmin usually has a background of related fields such as computer engineering, information technology, information systems, or even a trade school program [...] many sysadmins enter the field by self-taught." A recent survey from StackOverflow.com shows that sysadmins do not commonly have systemic training of programming—52% of the surveyed sysadmins learn programming on their own [146].

It is well recognized that sysadmins are very different from software developers for the lack of *understanding* of system internals and *debugging support* for production systems [69]. Unlike developers who implement the systems software and can debug the software programs, sysadmins did not write the code and are not familiar with the code; instead, they configure the systems based on documentation that is usually voluminous, obscure, and incomplete [88], which significantly impairs their understanding of the systems and the offered configurations. Moreover, sysadmins often cannot debug the systems in production in the same way as developers. Typically, if the sysadmins fail to address an issue, the issue will be escalated to the corresponding support engineers or developers who wrote the code.

In recent years, the concept of "DevOps" has emerged in industry with the core principles being the involvement of IT function in each phase of a system's development, heavy reliance on automation, and the application of engineering practices and tools to operations [29]. In the DevOps model, a team play both developer ("Dev") and operator ("Ops") roles. Despite being developed independently by Google, site reliability engineering can be viewed as a specific implementation of DevOps with some idiosyncratic extensions focusing on reliability of the services [29].

Although DevOps and site reliability engineering significantly change the landscape of system administration and operations by making operation an engineering discipline, they do not fundamentally address the dilemma faced by traditional sysadmins. Despite an engineering background, DevOps or site reliability engineers still need to manage code written by other teams/engineers. As a result, their understanding is limited [103]. Specially, with the massive scale of configurations across thousands of software components and stacks in cloud and datacenter systems, it is not possible to rely on manual efforts for managing configurations and ensuring their correctness. In fact, misconfigurations still remain a major cause of failures and outages in cloud-based companies that adopt DevOps and site reliability engineering such as Google and Facebook, as discussed in Chapter 1. Therefore, we believe that the principles and tool support presented in this dissertation are applicable to all different operation disciplines.

In this dissertation, we use the term "operators" to refer to people who play the role of system administration and operation and thus are responsible for configurations, despite their actual position, title, or education and work background.

Bibliography

[1] Configuration Datasets. https://github.com/tianyin/configuration_datasets.

[2] Cox: A configuration navigation tool and library for a thousand of knobs. https://github.com/tianyin/cox.

[3] Soot: a Java Optimization Framework. http://sable.github.io/soot/.

[4] The Apache Lucene Project. https://lucene.apache.org/.

[5] The LLVM Compiler Infrastructure Project. http://llvm.org/.

[6] AGGARWAL, B., BHAGWAN, R., DAS, T., ESWARAN, S., PADMANABHAN, V. N., AND VOELKER, G. M. NetPrints: Diagnosing Home Network Misconfigurations Using Shared Knowledge. In *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI'09)* (Boston, MA, 2009).

[7] AKEN, D. V., PAVLO, A., GORDON, G. J., AND ZHANG, B. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD'17)* (Chicago, IL, May 2017).

[8] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proceedings of the 14th USENIX Symposium on Networked System Design and Implementation (NSDI'17)* (Boston, MA, March 2017).

[9] ALVARO, P., ANDRUS, K., SANDEN, C., ROSENTHAL, C., BASIRI, A., AND HOCHSTEIN, L. Automating Failure Testing Research at Internet Scale. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)* (Santa Clara, CA, October 2016).

[10] ALVARO, P., ROSEN, J., AND HELLERSTEIN, J. M. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)* (Melbourne, Victoria, Australia, May 2015).

[11] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying Trends in Enterprise Data Protection Systems. In *Proceedings of 2015 USENIX Annual Technical Conference (ATC'15)* (Santa Clara, CA, July 2015).

[12] AMVROSIADIS, G., AND BHADKAMKAR, M. Getting Back Up: Understanding How Enterprise Data Backups Fail. In *Proceedings of 2016 USENIX Annual Technical Conference (ATC'16)* (Denver, CO, June 2016).

[13] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: Running Circles Around Storage Administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)* (Berkeley, CA, January 2002).

[14] ANDERSON, P., GOLDSACK, P., AND PATERSON, J. SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control. In *Proceedings of the 17th USENIX Large Installation System Administration Conference (LISA'03)* (San Diego, CA, October 2003).

[15] ANDERSON, P., AND SMITH, E. Configuration Tools: Working Together. In *Proceedings of the 19th Large Installation System Administration Conference (LISA'05)* (San Diego, CA, December 2005).

[16] ATTARIYAN, M., CHOW, M., AND FLINN, J. 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, October 2012).

[17] ATTARIYAN, M., AND FLINN, J. Using Causality to Diagnose Configuration Bugs. In *Proceedings of 2008 USENIX Annual Technical Conference (ATC'08)* (Boston, MA, June 2008).

[18] ATTARIYAN, M., AND FLINN, J. 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, October 2010).

[19] BARRETT, R., KANDOGAN, E., MAGLIO, P. P., HABER, E., TAKAYAMA, L. A., AND PRABAKER, M. Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW'04)* (Chicago, IL, November 2004).

[20] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines*, 2 ed. Morgan and Claypool Publishers, 2013.

[21] BASIRI, A., BEHNAM, N., DE ROOIJ, R., HOCHSTEIN, L., KOSEWSKI, L., REYNOLDS, J., AND ROSENTHAL, C. Chaos Engineering. *IEEE Software Magazine 33*, 3 (June 2016), 35–41.

[22] BAUER, L., GARRISS, S., AND REITER, M. K. Detecting and Resolving Policy Misconfigurations in Access-Control Systems. *ACM Transactions on Information and System Security (TISSEC) 14*, 1 (May 2011), 1–28.

[23] BECKETT, R., GUPTA, A., MAHAJAN, R., MILLSTEIN, T., AND WALKER, D. A General Approach to Network Configuration Verification. In *Proceedings of 2017 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'17)* (Los Angeles, CA, August 2016).

[24] BECKETT, R., MAHAJAN, R., MILLSTEIN, T., PADHYE, J., AND WALKER, D. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of 2016 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'16)* (Florianópolis, Brazil, August 2016).

[25] BECKETT, R., MAHAJAN, R., MILLSTEIN, T., PADHYE, J., AND WALKER, D. Network Configuration Synthesis with Abstract Topologies. In *Proceedings of the 38th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)* (Barcelona, Spain, June 2017).

[26] BEHRANG, F., COHEN, M. B., AND ORSO, A. Users Beware: Preference Inconsistencies Ahead. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy, August 2015).

[27] BENSON, T., AKELLA, A., AND MALTZ, D. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI'09)* (Boston, MA, 2009).

[28] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., HENRI-GROS, C., KAMSKY, A., MCPEAK, S., AND ENGLER, D. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM 53*, 2 (February 2010), 66–75.

[29] BEYER, B., JONES, C., PETOFF, J., AND MURPHY, N. R. *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media Inc., 2016.

[30] BIANCHINI, R., MARTIN, R. P., NAGARAJA, K., NGUYEN, T. D., AND OLIVEIRA, F. Human-Aware Computer System Design. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)* (June 2005).

[31] BORDER, C., AND BEGNUM, K. Educating System Administrators. *USENIX ;login: 39*, 5 (October 2014), 36–39.

[32] BRODKIN, J. Why Gmail went down: Google misconfigured load balancing servers. http://arstechnica.com/information-technology/2012/12/why-gmail-went-down-google-misconfigured-chromes-sync-server/.

[33] BROWN, A. B., AND PATTERSON, D. A. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 USENIX Annual Technical Conference (ATC'03)* (San Antonio, TX, June 2003).

[34] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation (OSDI'06)* (Seattle, WA, November 2006).

[35] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, December 2004).

[36] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)* (New Orleans, LA, February 1999).

[37] CHASE, J. S., ANDERSON, D. C., THAKAR, P. N., VAHDAT, A. M., AND DOYLE, R. P. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)* (Chateau Lake Louise, Banff, Canada, October 2001).

[38] CHAUDHURI, S., AND WEIKUM, G. Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)* (Cairo, Egypt, August 2000).

[39] CHEN, W., WU, H., WEI, J., ZHONG, H., AND HUANG, T. Determine Configuration Entry Correlations for Web Application Systems. In *Proceedings of the 40th IEEE Computer Software and Applications Conference* (Georgia, GA, June 2016).

[40] CHOW, M., VEERARAGHAVAN, K., CAFARELLA, M., AND FLINN, J. DQBarge: Improving data-quality tradeoffs in large-scale Internet services. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Savannah, GA, November 2016).

[41] DAS, T., BHAGWAN, R., AND NALDURG, P. Baaz: A System for Detecting Access Control Misconfigurations. In *Proceedings of the 19th USENIX Security Symposium (USENIX Security'10)* (Washington, DC, August 2010).

[42] Delaet, T., Joosen, W., and Vanbrabant, B. A Survey of System Configuration Tools. In *Proceedings of the 24th Large Installation System Administration Conference (LISA'10)* (San Jose, CA, November 2010).

[43] Dong, Z., Andrzejak, A., Lo, D., and Costa, D. ORPLocator: Identifying Read Points of Configuration Options via Static Analysis. In *Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering (ISSRE'16)* (Ottawa, ON, Canada, October 2016).

[44] Dong, Z., Andrzejak, A., and Shao, K. Practical and Accurate Pinpointing of Configuration Errors using Static Analysis. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution* (Bremen, Germany, September 2015).

[45] Duan, S., Thummala, V., and Babu, S. Tuning Database Configuration Parameters with iTuned. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)* (Lyon, France, August 2009).

[46] Dumlu, E., Yilmaz, C., Cohen, M. B., and Porter, A. Feedback Driven Adaptive Combinatorial Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11)* (Toronto, ON, Canada, July 2011).

[47] Enck, W., McDaniel, P., Sen, S., Sebos, P., Spoerel, S., Greenberg, A., Rao, S., and Aiello, W. Configuration Management at Massive Scale: System Design and Experience. In *Proceedings of 2007 USENIX Annual Technical Conference (ATC'07)* (Santa Clara, CA, June 2007).

[48] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)* (Chateau Lake Louise, Banff, Canada, October 2001).

[49] Fang, L., Nguyen, K., Xu, G., Demsky, B., and Lu, S. Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs. In *Proceedings of the 25th Symposium on Operating System Principles (SOSP'15)* (Monterey, CA, October 2015).

[50] Feamster, N., and Balakrishnan, H. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)* (Boston, MA, May 2005).

[51] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. A General Approach to Network Configuration Analysis. In *Proceedings of the 12th USENIX Symposium on*

Networked System Design and Implementation (NSDI'15) (Oakland, CA, May 2015).

[52] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (Vancouver, BC, Canada, October 2010).

[53] FU, W., PERERA, R., ANDERSON, P., AND CHENEY, J. μPuppet: A Declarative Subset of the Puppet Configuration Language. In *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP'17)* (Barcelona, Spain, June 2017).

[54] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure. In *Proceedings of 2016 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'16)* (Florianópolis, Brazil, August 2016).

[55] GRAY, J. Why Do Computers Stop and What Can Be Done About It? *Tandem Technical Report 85.7* (June 1985).

[56] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)* (Seattle, WA, November 2014).

[57] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. 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, October 2016).

[58] HABER, E. M., AND BAILEY, J. Design Guidelines for System Administration Tools Developed through Ethnographic Field Study. In *Proceedings of the 2007 ACM Conference on Human Interfaces to the Management of Information Technology (CHIMIT'07)* (Cambridge, MA, March 2007).

[59] HADOOP ISSUE #134. JobTracker trapped in a loop if it fails to localize a task. https://issues.apache.org/jira/browse/HADOOP-134.

[60] HADOOP ISSUE #2081. Configuration getInt, getLong, and getFloat replace invalid numbers with the default value. https://issues.apache.org/jira/browse/HADOOP-2081.

[61] HADOOP ISSUE #6578. Configuration should trim whitespace around a lot of value types. https://issues.apache.org/jira/browse/HADOOP-6578.

[62] HADOOP-USER MAILING LIST ARCHIVES. Compression codec com.hadoop.compression.lzo.LzoCodec not found. http://goo.gl/N9XFvt.

[63] HBASE ISSUE #6973. Trim trailing whitespace from configuration values. https://issues.apache.org/jira/browse/HBASE-6973.

[64] HDFS ISSUE 5872#. Validate configuration of dfs.http.policy. https://issues.apache.org/jira/browse/HDFS-5872.

[65] HDFS ISSUE #7726. Parse and check the configuration settings of edit log to prevent runtime errors. https://issues.apache.org/jira/browse/HDFS-7726.

[66] HDFS ISSUE #7727. Check and verify the auto-fence settings to prevent failures of auto-failover. https://issues.apache.org/jira/browse/HDFS-7727.

[67] HERODOTOU, H., AND BABU, S. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. In *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB'11)* (Seattle, WA, August 2011).

[68] HOFFMANN, H., SIDIROGLOU, S., CARBIN, M., MISAILOVIC, S., AGARWAL, A., AND RINARD, M. Dynamic Knobs for Responsive Power-Aware Computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)* (Newport Beach, CA, March 2011).

[69] HREBEC, D. G., AND STIBER, M. A Survey of System Administrator Mental Models and Situation Awareness. In *Proceedings of the 2001 Special Interest Group on Computer Personnel Research Annual Conference (SIGCPR'01)* (Vancouver, BC, Canada, 2001).

[70] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure Coding in Windows Azure Storage. In *Proceedings of 2012 USENIX Annual Technical Conference (ATC'12)* (Boston, MA, June 2012).

[71] HUANG, P., BOLOSKY, W. J., SIGH, A., AND ZHOU, Y. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proceedings of the 10th ACM European Conference in Computer Systems (EuroSys'15)* (Bordeaux, France, April 2015).

[72] HUANG, P., GUO, C., ZHOU, L., LORCH, J. R., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)* (Whistler, British Columbia, Canada, May 2017).

[73] HUANG, P., XU, T., JIN, X., AND ZHOU, Y. DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior. In *Proceedings of the 14th International Conference on Mobile Systems, Applications, and Services (MobiSys'16)* (Singapore, Singapore, June 2016).

[74] HUBAUX, A., XIONG, Y., AND CZARNECKI, K. A User Survey of Configuration Challenges in Linux and eCos. In *Proceedings of 6th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12)* (Leipzig, Germany, January 2012).

[75] HWAN, C., KIM, P., MARINOV, D., KHURSHID, S., AND BATORY, D. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)* (Saint Petersburg, Russia, August 2013).

[76] JHA, A. K., LEE, S., AND LEE, W. J. Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)* (Buenos Aires, Argentina, May 2017).

[77] JIANG, W., HU, C., PASUPATHY, S., KANEVSKY, A., LI, Z., AND ZHOU, Y. Understanding Customer Problem Troubleshooting from Storage System Logs. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)* (San Francisco, CA, February 2009).

[78] JIN, D., COHEN, M. B., QU, X., AND ROBINSON, B. PrefFinder: Getting the Right Preference in Configurable Software Systems. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE'14)* (Västerås, Sweden, September 2014).

[79] JIN, D., QU, X., COHEN, M. B., AND ROBINSON, B. Configurations Everywhere: Implications for Testing and Debugging in Practice. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)* (Hyderabad, India, 2014).

[80] JIN, X., HUANG, P., XU, T., AND ZHOU, Y. NChecker: Saving Mobile App Developers from Network Disruptions. In *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys'16)* (London, UK, April 2016).

[81] KANDOGAN, E., AND HABER, E. M. Security Administration Tools and Practices. *Security and Usability, O'Reilly Media, Inc.* (August 2005).

[82] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)* (San Francisco, CA, March 2004).

[83] KELLER, L., UPADHYAYA, P., AND CANDEA, G. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)* (Anchorage, AK, June 2008).

[84] KENDRICK, S. What Takes Us Down? *USENIX ;login: 37*, 5 (October 2012), 37–45.

[85] KICIMAN, E., AND WANG, Y.-M. Discovering Correctness Constraints for Self-Management of System Configuration. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)* (New York, NY, May 2004).

[86] KUSHMAN, N., AND KATABI, D. Enabling Configuration-Independent Automation by Non-Expert Users. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (Vancouver, BC, Canada, October 2010).

[87] LABOVITZ, C., AHUJA, A., AND JAHANIAN, F. Experimental Study of Internet Stability and Backbone Failures. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS'99)* (June 1999).

[88] LAMPSON, B. W. Computer Security in the Real World. *IEEE Computer 37*, 6 (June 2004), 37–46.

[89] LARSSON, M., AND CRNKOVIC, I. Configuration Management for Component-based Systems. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)* (Toronto, Ontario, Canada, May 2001).

[90] LEESATAPORNWONGSA, T., AND GUNAWI, H. S. The Case for Drill-Ready Cloud Computing. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)* (Seattle, WA, November 2014).

[91] LEESATAPORNWONGSA, T., STUARDO, C. A., SUMINTO, R. O., KE, H., LUKMAN, J. F., AND GUNAWI, H. S. Scalability Bugs: When 100-Node Testing is Not Enough. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)* (Whistler, British Columbia, Canada, May 2017).

[92] LI, W., LI, S., LIAO, X., XU, X., ZHOU, S., AND JIA, Z. ConfTest: Generating Comprehensive Misconfiguration for System Reaction Ability Evaluation. In *Proceedings of the 21th International Conference on Evaluation and Assessment in Software Engineering* (Karlskrona, Sweden, June 2017).

[93] LI, Z., WANG, W., XU, T., ZHONG, X., LI, X.-Y., LIU, Y., WILSON, C., AND ZHAO, B. Y. Exploring Cross-Application Cellular Traffic Optimization with Baidu TrafficGuard. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)* (Santa Clara, CA, March 2016).

[94] LIANG, L. Y. Linkedin.com inaccessible on Thursday because of server misconfiguration. http://www.straitstimes.com/breaking-news/singapore/story/linkedincom-inaccessible-thursday-because-server-misconfiguration-2013.

[95] LUTTERKORT, D. Augeas—a configuration API. In *Proceedings of the 10th Linux Symposium* (Ottawa, Ontario, Canada, June 2008).

[96] MAHAJAN, R., WETHERALL, D., AND ANDERSON, T. Understanding BGP Misconfiguration. In *Proceedings of 2002 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'02)* (Pittsburgh, PA, August 2002).

[97] MAURER, B. Fail at Scale: Reliability in the Face of Rapid Change. *Communications of the ACM 58*, 11 (November 2015), 44–49.

[98] MAYHEW, D. J. *Principles and Guidelines in Software User Interface Design*. Prentice Hall, October 1991.

[99] MEINICKE, J., WONG, C.-P., KÄSTNER, C., THÜM, T., AND SAAKE, G. On Essential Configuration Complexity: Measuring Interations in Highly-Configurable Systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)* (Singapore, Singapore, September 2016).

[100] MICHEL, R., HUBAUX, A., GANESH, V., AND HEYMANS, P. An SMT-based Approach to Automated Configuration. In *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT'12)* (Manchester, UK, June 2012).

[101] MICKENS, J., SZUMMER, M., AND NARAYANAN, D. Snitch: Interactive Decision Trees for Troubleshooting Misconfigurations. In *Proceedings of the 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SYSML'07)* (April 2007).

[102] MILLER, B. P., KOSKI, D., LEE, C. P., MAGANTY, V., MURTHY, R., NATARAJAN, A., AND STEIDL, J. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Tech. Rep. 1268, University of Wisconsin-Madison, Computer Sciences Department, April 1995.

[103] MOGUL, J. C., ISAACS, R., AND WELCH, B. Thinking about Availability in Large Service Infrastructures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)* (Whistler, British Columbia, Canada, May 2017).

[104] MOSKOWITZ, A. Software Testing for Sysadmin Programs. *USENIX ;login: 40*, 2 (April 2015), 37–45.

[105] MYSQL BUG #74720. No warn/error message if "log-error" is misconfigured (causing latent log loss). http://bugs.mysql.com/bug.php?id=74720.

[106] MYSQL BUG #75645. Runtime Error Caused by Misconfigured BackupDataDir. http://bugs.mysql.com/bug.php?id=75645.

[107] NADI, S., BERGER, T., KÄSTNER, C., AND CZARNECKI, K. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)* (Hyderabad, India, 2014).

[108] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, December 2004).

[109] NAIR, V., MENZIES, T., SIEGMUND, N., AND APEL, S. Using Bad Learners to Find Good Configurations. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17)* (Paderborn, Germany, September 2017).

[110] NARAIN, S., LEVIN, G., KAUL, V., AND MALIK, S. Declarative Infrastructure Configuration Synthesis and Debugging. *Journal of Network and System Management 16*, 3 (2008), 235–258.

[111] NORMAN, D. A. Design Rules Based on Analyses of Human Error. *Communications of the ACM 26*, 4 (April 1983), 254–258.

[112] OLIVEIRA, F., NAGARAJA, K., BACHWANI, R., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Validating Database System Administration. In *Proceedings of 2006 USENIX Annual Technical Conference (ATC'06)* (May 2006).

[113] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Seattle, WA, March 2003).

[114] OSOGAMI, T., AND ITOKO, T. Finding Probably Better System Configurations Quickly. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance'06)* (June 2006).

[115] PALATIN, N., LEIZAROWITZ, A., SCHUSTER, A., AND WOLFF, R. Mining for Misconfigured Machines in Grid Systems. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)* (Philadelphia, PA, August 2006).

[116] PATTERSON, D., BROWN, A., BROADWELL, P., CANDEA, G., CHEN, M., CUTLER, J., ENRIQUEZ, P., FOX, A., KICIMAN, E., MERZBACHER, M., OPPENHEIMER, D., SASTRY, N., TETZLAFF, W., TRAUPMAN, J., AND TREUHAFT, N. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Tech. Rep. UCB//CSD-02-1175, University of California Berkeley, March 2002.

[117] PERROW, C. *Normal Accidents: Living with High-Risk Technologies*. Basic Books, 1984.

[118] POTHARAJU, R., CHAN, J., HU, L., NITA-ROTARU, C., WANG, M., ZHANG, L., AND JAIN, N. ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'15)* (Kohala Coast, HI, August 2015).

[119] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. RX: Treating Bugs As Allergies—A Safe Method to Survive Software Failure. In *Proceedings of the 20th Symposium on Operating System Principles (SOSP'05)* (Brighton, United Kingdom, October 2005).

[120] QU, X., ACHARYA, M., AND ROBINSON, B. Impact Analysis of Configuration Changes for Test Case Selection. In *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering (ISSRE'11)* (Hiroshima, Japan, November 2011).

[121] QU, X., COHEN, M. B., AND ROTHERMEL, G. Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'08)* (Seattle, WA, July 2008).

[122] RAAB, M., AND BARANY, G. Challenges in Validating FLOSS Configuration. In *Proceedings of the 13th International Conference on Open Source Systems* (Buenos Aires, Argentina, May 2017).

[123] RABKIN, A., AND KATZ, R. Precomputing Possible Configuration Error Diagnosis. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)* (Lawrence, KS, November 2011).

[124] RABKIN, A., AND KATZ, R. Static Extraction of Program Configuration Options. In *Proceedings of the 33th International Conference on Software Engineering (ICSE'11)* (Honolulu, HI, May 2011).

[125] RABKIN, A., AND KATZ, R. How Hadoop Clusters Break. *IEEE Software Magazine 30*, 4 (July 2013), 88–94.

[126] RAMACHANDRAN, V., GUPTA, M., SETHI, M., AND CHOWDHURY, S. R. Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-tiered Enterprise Applications. In *Proceedings of the 6th International Conference on Autonomic Computing and Communications (ICAC'09)* (Barcelona, Spain, June 2009).

[127] REASON, J. *Human Error*. Cambridge University Press, October 1990.

[128] REISNER, E., SONG, C., MA, K.-K., FOSTER, J. S., AND PORTER, A. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32th International Conference on Software Engineering (ICSE'10)* (Cape Town, South Africa, May 2010).

[129] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., AND WILLIAM S. BEEBEE, J. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, December 2004).

[130] ROBBINS, J., KRISHNAN, K., ALLSPAW, J., AND LIMONCELLI, T. Resilience Engineering: Learning to Embrace Failure. *ACM Queue 10*, 9 (September 2012), 1–9.

[131] SANTOLUCITO, M., ZHAI, E., AND PISKAC, R. Probabilistic Automated Language Learning for Configuration Files. In *28th International Conference on Computer Aided Verification (CAV'16)* (Toronto, Canada, July 2016).

[132] SAYAGH, M., KERZAZI, N., AND ADAMS, B. On Cross-stack Configuration Errors. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)* (Buenos Aires, Argentina, May 2017).

[133] SAYYAD, A. S., INGRAM, J., MENZIES, T., AND AMMAR, H. Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)* (Silicon Valley, CA, November 2013).

[134] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., and Young, M. Machine Learning: The High-Interest Credit Card of Technical Debt. In *Proceedings of the 1st Workshop on Software Engineering for Machine Learning* (Montreal, Canada, December 2014).

[135] Search Engine Watch. How Much is a Google Top Spot Worth? http://searchenginewatch.com/article/2050861/How-Much-is-a-Google-Top-Spot-Worth.

[136] Search Engine Watch. 53% of Organic Search Clicks Go to First Link. http://searchenginewatch.com/article/2050861/How-Much-is-a-Google-Top-Spot-Worth.

[137] Shambaugh, R., Weiss, A., and Guha, A. Rehearsal: A Configuration Verification Tool for Puppet. In *Proceedings of the 37th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)* (Santa Barbara, CA, June 2016).

[138] Sherman, A., Lisiecki, P., Berkheimer, A., and Wein, J. ACMS: Akamai Configuration Management System. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)* (Boston, MA, May 2005).

[139] Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy, August 2015).

[140] Siewiorek, D. P., and Swarz, R. S. *Reliable Computer Systems: Design and Evaluation*, 3 ed. A K Peters, Ltd, 1998.

[141] Song, C., Porter, A., and Foster, J. S. iTree: Efficiently Discovering High-Coverage Configuration Using Interaction Trees. *IEEE Transactions on Software Engineering (TSE) 40*, 3 (March 2014), 251–265.

[142] Squid Bug #1703. diskd related 100% CPU after 'squid -k rotate'. http://bugs.squid-cache.org/show_bug.cgi?id=1703.

[143] Squid Bug #4186. The mail notification feature is buggy and does not deal with configuration errors. http://bugs.squid-cache.org/show_bug.cgi?id=4186.

[144] Sridharan, M., Fink, S. J., and Bodík, R. Thin Slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)* (San Diego, CA, June 2007).

[145] SRIKANTH, H., COHEN, M. B., AND QU, X. Reducing Field Failures in System Configurable Software: Cost-Based Prioritization. In *Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering (ISSRE'09)* (Mysuru, Karnataka, India, November 2009).

[146] STACKOVERFLOW. 2015 Developer Survey. http://stackoverflow.com/research/developer-survey-2015#profile-education, 2015.

[147] STACKOVERFLOW QUESTION #21253299. Hadoop sshfence (permission denied). http://stackoverflow.com/questions/21253299/hadoop-sshfence-permission-denied.

[148] STACKOVERFLOW QUESTION #7732983. Core dump file is not generated. http://stackoverflow.com/questions/7732983/core-dump-file-is-not-generated.

[149] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)* (Stevenson, WA, October 2007).

[150] SU, Y.-Y., AND FLINN, J. Automatically Generating Predicates and Solutions for Configuration Troubleshooting. In *Proceedings of 2009 USENIX Annual Technical Conference (ATC'09)* (San Diego, CA, June 2009).

[151] SUNG, Y.-W. E., TIE, X., WONG, S. H., AND ZENG, H. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of 2016 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'16)* (Florianópolis, Brazil, August 2016).

[152] SVERDLIK, Y. Microsoft: Misconfigured Network Device Led to Azure Outage. http://www.datacenterdynamics.com/focus/archive/2012/07/microsoft-misconfigured-network-device-led-azure-outage, 2012.

[153] TAKAYAMA, L., AND KANDOGAN, E. Trust as an Underlying Factor of System Administrator Interface Choice. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems (CHI EA'06)* (Montréal, Québec, Canada, April 2006).

[154] TAMURA, G., CASALLAS, R., CLEVE, A., AND DUCHIEN, L. QoS Contract Preservation through Dynamic Reconfiguration: A Formal Semantics Approach. *Science of Computer Programming 94*, 3 (November 2014), 301–332.

[155] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating System Principles (SOSP'15)* (Monterey, CA, October 2015).

[156] THE ASSOCIATION OF SUPPORT PROFESSIONALS. *Technical Support Cost Ratios* (2000).

[157] THE AVAILABILITY DIGEST. Poor Documentation Snags Google. http://www.availabilitydigest.com/public_articles/0504/google_power_out.pdf.

[158] THOMAS, K. Thanks, Amazon: The Cloud Crash Reveals Your Importance. http://www.pcworld.com/article/226033/thanks_amazon_for_making_possible_much_of_the_internet.html.

[159] TSEITLIN, A. The Antifragile Organization. *Communications of the ACM 56*, 8 (August 2013), 40–44.

[160] VECCHIATO, D., VIEIRA, M., AND MARTINS, E. The Perils of Android Security Configuration. *IEEE Computer 49*, 6 (June 2016), 15–21.

[161] VEERARAGHAVAN, K., MEZA, J., CHOU, D., KIM, W., MARGULIS, S., MICHELSON, S., NISHTALA, R., OBENSHAIN, D., PERELMAN, D., AND SONG, Y. J. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)* (Savannah, GA, November 2016).

[162] VELASQUEZ, N. F., WEISBAND, S., AND DURCIKOVA, A. Designing Tools for System Administrators: An Empirical Test of the Integrated User Satisfaction Model. In *Proceedings of the 22nd Large Installation System Administration Conference (LISA'08)* (San Diego, CA, November 2008).

[163] VERBOWSKI, C., KICIMAN, E., KUMAR, A., DANIELS, B., LU, S., LEE, J., WANG, Y.-M., AND ROUSSEV, R. Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation (OSDI'06)* (Seattle, WA, November 2006).

[164] VERBOWSKI, C., LEE, J., LIU, X., ROUSSEV, R., AND WANG, Y.-M. LiveOps: Systems Management as a Service. In *Proceedings of the 20th Large Installation System Administration Conference (LISA'06)* (December 2006).

[165] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, December 2004).

[166] WANG, T., HARMAN, M., JIA, Y., AND KRINKE, J. Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and*

the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13) (Saint Petersburg, Russia, August 2013).

[167] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Broomfield, CO, October 2014).

[168] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)* (San Diego, CA, October 2003).

[169] WELSH, M. What I wish systems researchers would work on. http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html.

[170] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, December 2004).

[171] XI, B., LIU, Z., RAGHAVACHARI, M., XIA, C. H., AND ZHANG, L. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *Proceedings of the 13th International World Wide Web Conference (WWW'04)* (May 2004).

[172] XIA, X., LO, D., QIU, W., WANG, X., AND ZHOU, B. Automated Configuration Bug Report Prediction Using Text Mining. In *Proceedings of the 38th IEEE Computer Software and Applications Conference* (Västerås, Sweden, June 2014).

[173] XIONG, Y., ZHANG, H., HUBAUX, A., SHE, S., WANG, J., AND CZARNECKI, K. Range Fixes: Interactive Error Resolution for Software Configuration. *IEEE Transactions on Software Engineering (TSE) 41*, 6 (June 2015), 603–619.

[174] XU, B., LO, D., XIA, X., SUREKA, A., AND LI, S. EFSPredictor: Predicting Configuration Bugs With Ensemble Feature Selection. In *Proceedings of the 22nd Asia-Pacific Software Engineering Conference* (New Delhi, India, December 2015).

[175] XU, T., JIN, L., FAN, X., ZHOU, Y., PASUPATHY, S., AND TALWADKER, R. Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy, August 2015).

[176] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Savannah, GA, November 2016).

[177] XU, T., NAING, H. M., LU, L., AND ZHOU, Y. How Do System Administrators Resolve Access-Denied Issues in the Real World? In *Proceedings of the 35th Annual CHI Conference on Human Factors in Computing Systems (CHI'17)* (Denver, CO, May 2017).

[178] XU, T., PANDEY, V., AND KLEMMER, S. An HCI View of Configuration Problems. *CoRR abs/1601.01747* (January 2016).

[179] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)* (Farmington, PA, November 2013).

[180] XU, T., AND ZHOU, Y. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR) 47*, 4 (July 2015).

[181] XU, X., LI, S., GUO, Y., DONG, W., LI, W., AND LIAO, X. Automatic Type Inference for Proactive Misconfiguration Prevention. In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering* (Pittsburgh, PA, June 2017).

[182] YARN ISSUE #2166. Timelineserver should validate that yarn.timeline-service.leveldb-timeline-store.ttl-interval-ms is greater than zero when level db is for timeline store. https://issues.apache.org/jira/browse/YARN-2166.

[183] YE, T., AND KALYANARAMAN, S. A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'03)* (June 2003).

[184] YILMAZ, C., COHEN, M. B., AND PORTER, A. A. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. *IEEE Transactions on Software Engineering (TSE) 32*, 1 (January 2006), 1–15.

[185] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (Cascais, Portugal, October 2011).

[186] YUAN, C., LAO, N., WEN, J.-R., LI, J., ZHANG, Z., WANG, Y.-M., AND MA, W.-Y. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the 1st EuroSys Conference (EuroSys'06)* (Leuven, Belgium, April 2006).

[187] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. 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, October 2014).

[188] YUAN, D., PARK, S., HUANG, P., LIU, Y., LEE, M. M., TANG, X., ZHOU, Y., AND SAVAGE, S. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (Hollywood, CA, October 2012).

[189] YUAN, D., XIE, Y., PANIGRAHY, R., YANG, J., VERBOWSKI, C., AND KUMAR, A. Context-based Online Configuration Error Detection. In *Proceedings of 2011 USENIX Annual Technical Conference (ATC'11)* (Portland, OR, June 2011).

[190] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)* (Salt Lake City, UT, March 2014).

[191] ZHANG, S., AND ERNST, M. D. Automated Diagnosis of Software ConïňĄguration Errors. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)* (San Francisco, CA, May 2013).

[192] ZHANG, S., AND ERNST, M. D. Which Configuration Option Should I Change? In *Proceedings of the 36th Internationl Conference on Software Engineering (ICSE'14)* (Hyderabad, India, May 2014).

[193] ZHANG, S., AND ERNST, M. D. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)* (Baltimore, MD, July 2015).

[194] ZHENG, W., BIANCHINI, R., AND NGUYEN, T. D. Automatic Configuration of Internet Services. In *Proceedings of the 2nd EuroSys Conference (EuroSys'07)* (Lisbon, Portugal, March 2007).

[195] ZHENG, W., BIANCHINI, R., AND NGUYEN, T. D. MassConf: Automatic Configuration Tuning By Leveraging User Community Information. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering* (Karlsruhe, Germany, March 2011).

[196] ZHOU, S., LI, S., LIU, X., XU, X., ZHENG, S., LIAO, X., AND XIONG, Y. Easier Said Than Done: Diagnosing Misconfiguration via Configuration Constraints

Analysis: A Study of the Variance of Configuration Constraints in Source Code. In *Proceedings of the 21th International Conference on Evaluation and Assessment in Software Engineering* (Karlskrona, Sweden, June 2017).

[197] ZHOU, S., LIU, X., LI, S., DONG, W., LIAO, X., AND XIONG, Y. ConfMapper: Automated Variable Finding for Configuration Items in Source Code. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability, and Security* (Vienna, Austria, August 2016).

[198] ZHU, Y., LIU, J., GUO, M., BAO, Y., SONG, K., AND LIU, Z. BestConfig: Tapping the Performance Potential of Systems via Configuration Adjustment. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC'17)* (Santa Clara, CA, September 2017).