An Empirical Study on Configuration Errors in Commercial and Open Source Systems

Zuoning Yin*, Xiao Ma*, Jing Zheng†, Yuanyuan Zhou†, Lakshmi N. Bairavasundaram‡, and Shankar Pasupathy‡
*Univ. of Illinois at Urbana-Champaign, †Univ. of California, San Diego, ‡NetApp, Inc.

ABSTRACT

Configuration errors (i.e., misconfigurations) are among the dominant causes of system failures. Their importance has inspired many research efforts on detecting, diagnosing, and fixing misconfigurations; such research would benefit greatly from a real-world characteristic study on misconfigurations. Unfortunately, few such studies have been conducted in the past, primarily because historical misconfigurations usually have not been recorded rigorously in databases.

In this work, we undertake one of the first attempts to conduct a real-world misconfiguration characteristic study. We study a total of 546 real world misconfigurations, including 309 misconfigurations from a commercial storage system deployed at thousands of customers, and 237 from four widely used open source systems (CentOS, MySQL, Apache HTTP Server, and OpenLDAP). Some of our major findings include: (1) A majority of misconfigurations (70.0%∼85.5%) are due to mistakes in setting configuration parameters; however, a significant number of misconfigurations are due to compatibility issues or component configurations (i.e., not parameter-related). (2) 38.1%∼53.7% of parameter mistakes are caused by illegal parameters that clearly violate some format or rules, motivating the use of an automatic configuration checker to detect these misconfigurations. (3) A significant percentage (12.2%∼29.7%) of parameter-based mistakes are due to inconsistencies between different parameter values. (4) 21.7%∼57.3% of the misconfigurations involve configurations external to the examined system, some even on entirely different hosts. (5) A significant portion of misconfigurations can cause hard-to-diagnose failures, such as crashes, hangs, or severe performance degradation, indicating that systems should be better-equipped to handle misconfigurations.

Categories and Subject Descriptors: D.4.5 [Operating Systems]: Reliability

General Terms: Reliability, Management

Keywords: Misconfigurations, characteristic study

SOSP '11, October 23-26, 2011, Cascais, Portugal.

1. INTRODUCTION

1.1 Motivation

Configuration errors (i.e., misconfigurations) have a great impact on system availability. For example, a recent misconfiguration at Facebook prevented its 500 million users from accessing the website for several hours [15]. Last year, a misconfiguration brought down the entire ".se" domain for more than an hour [6], affecting almost 1 million hosts.

Not only do misconfigurations have high impact, they are also prevalent. Gray's pioneering paper on system faults [11] stated that administrator errors were responsible for 42% of system failures in high-end mainframes. Similarly, Patterson et al. [30] observed that more than 50% of failures were due to operator errors in telephone networks and Internet systems. Studies have also observed that a majority of operator errors (or administrator errors) are misconfigurations [23, 29]. Further, of the issues reported in COMP-A's[1] customer-support database (used in this study), around 27% are labeled as configuration-related (as shown later in Figure 1(a) in Section 3). This percentage is second only to hardware failures and is much bigger than that of software bugs.

Moreover, configuration errors are also expensive to troubleshoot. Kappor [16] found that 17% of the total cost of ownership of today's desktop computers goes toward technical support, and a large fraction of that is troubleshooting misconfigurations.

Given the data on the prevalence and impact of misconfigurations, several recent research efforts [3, 17, 18, 35, 38, 41] have proposed ideas to detect, diagnose, and automatically fix misconfigurations. For example, PeerPressure [38] uses statistics methods on a large set of configurations to identify single configuration parameter errors. Chronus [41] periodically checkpoints disk state and automatically searches for configuration changes that may have caused the misconfiguration. ConfAid [3] uses data flow analysis to trace the configuration error back to a particular configuration entry. AutoBash [35] leverages a speculative OS kernel to automatically try out fixes from a solution database in order to find a proper solution for a configuration problem. Further, ConfErr [17] provides a useful framework with which users can inject configuration errors of three types: typos, structural mistakes, and semantic mistakes. In addition to research efforts, various tools are available to aid users in managing

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

Major Findings on Prevalence and Severity of Configuration Issues (Section 3)
Similar to results from previous studies [11, 29, 30], data from COMP-A shows that a significant portion (27%) of customer cases are related to configuration issues.
Configuration issues cause the largest percentage (31%) of high-severity support requests.
Major Findings on Misconfiguration Types (Section 4)
Configuration-parameter mistakes account for the majority (70.0%∼85.5%) of the examined misconfigurations.
However, a significant portion (14.5%∼30.0%) of the examined misconfigurations are caused by software compatibility issues and component configuration, which are not well addressed in literature.
38.1%∼53.7% of parameter misconfigurations are caused by illegal parameters that violate formats or semantic rules defined by the system, and can be potentially detected by checkers that inspect against these rules.
A significant portion (12.2%∼29.7%) of parameter mistakes are due to value-based inconsistency, calling for an inconsistency checker or a better configuration design that does not require users to worry about such error-prone consistency constraints.
Although most misconfigurations are located within each examined system, still a significant portion (21.7%∼57.3%) involve configurations beyond the system itself or span over multiple hosts.
Major Findings on System Reactions to Misconfigurations (Section 5)
Only 7.2%∼15.5% of the studied misconfiguration problems provide explicit messages that pinpoint the configuration error.
Some misconfigurations have caused the systems to crash, hang or have severe performance degradation, making failure diagnosis a challenging task.
Messages that pinpoint configuration errors can shorten the diagnosis time by 3 to 13 times as compared to the cases with ambiguous messages or by 1.2 to 14.5 times as compared to cases with no messages.
Major Findings on Causes of Misconfigurations (Section 6)
The majority of misconfigurations are related to first-time use of desired functionality. For more complex systems, a significant percentage (16.7%∼32.4%) of misconfigurations were introduced into systems that used to work.
By looking into the 100 used-to-work cases (32.4% of the total) at COMP-A, 46% of them are attributed to configuration parameter changes due to routine maintenance, configuring for new functionality, system outages, etc, and can benefit from tracking configuration changes. The remainder are caused by non-parameter related issues such as hardware changes (18%), external environmental changes (8%), resource exhaustion (14%), and software upgrades(14%).
Major Findings on Impact of Misconfigurations (Section 7)
Although most studied misconfiguration cases only lead to partial unavailability of the system, 16.1%∼47.3% of them make the systems to be fully unavailable or cause severe performance degradation.

Table 1: Major findings on misconfiguration characteristics. Please take our methodology into consideration when you interpret and draw any conclusions.

system configuration; for example, storage systems have provisioning tools [13, 14, 25, 26], misconfiguration-detection tools [24], and upgrade assistants that check for compatibility issues [24]. The above research directions and tools would benefit greatly from a characteristic study of real-world misconfigurations. Moreover, understanding the major types and root causes of misconfigurations may help guide developers to better design configuration logic and requirements, and testers to better verify user interfaces, thereby reducing the likelihood of configuration mistakes by users.

Unfortunately, in comparison to software bugs that have well-maintained bug databases and have benefited from many software bug characteristic studies [5, 19, 36, 37], a misconfiguration characteristic study is much harder, mainly because historical misconfigurations usually have not been recorded rigorously in databases. For example, developers record information about the context in the code for bugs, the causes of bugs, and how they were fixed; they also focus on eliminating or coalescing duplicate bug reports. On the other hand, the description of misconfigurations is user-driven, the fixes may be recorded simply as pointers to manuals and best-practice documents, and there is no duplicate elimination. As a result, analyzing and understanding misconfigurations is a much harder, and more importantly, manual task.

1.2 Our Contributions

In this paper, we perform one of the first characteristic studies of real-world misconfigurations in both commercial and open-source systems, using a total of 546 misconfiguration cases. The commercial system is a storage system from COMP-A deployed at thousands of customers. It has a well-maintained customer-issues database. The open-source systems include widely used system software: CentOS, MySQL, Apache, and OpenLDAP. The misconfiguration issues we examine are primarily user-reported. Therefore, our study is a manual analysis of user descriptions of misconfigurations, aided by discussions with developers, support engineers, and system architects of these systems to ensure correct understanding of these cases. Our study was approximately 21 person-months of effort, excluding the help from several COMP-A engineers and open-source developers.

We study the types, patterns, causes, system reactions, and impact of misconfigurations:

- We examine the prevalence and reported severity of configuration issues (includes, but not limited to misconfigurations) as compared to other support issues in COMP-A's customer-issues database.
- We develop a simple taxonomy of misconfiguration types: *parameter*, *compatibility*, and *component*, and identify the prevalence of each type. Given the prevalence of parameter-based misconfigurations, we further analyze its types and observable patterns.
- We identify how systems react to misconfigurations: whether error messages are provided, whether systems experience failures or severe performance issues, etc. Given that error messages are important for diagnosis and fixes, we also investigate the relationship between message clarity and diagnosis time.

- We study the frequency of different causes of misconfigurations such as first-time use, software upgrades, hardware changes, etc.
- Finally, we examine the impact of misconfigurations, including the impact on system availability and performance.

The major findings of the study are summarized in Table 1. While we believe that the misconfiguration cases we examined are fairly representative of misconfigurations in large system software, we do not intend to draw any general conclusions about all applications. In particular, we remind readers that all of the characteristics and findings in this study should be taken with the specific system types and our methodology in mind (discussed in Section 2).

We will release our open-source misconfiguration cases to share with the research community.

2. METHODOLOGY

This section describes our methodology for analyzing misconfigurations. There are unique challenges in obtaining and analyzing a large set of real-world misconfigurations. Historically, unlike bugs that usually have Bugzillas as repositories, misconfigurations are not recorded rigorously. Much of the information is in the form of unstructured textual descriptions and there is no systematic way to report misconfiguration cases. Therefore, in order to overcome these challenges, we manually analyzed reported misconfiguration cases by studying manuals, instructions, source code, and knowledge bases of each system. For some hard cases, we contacted the corresponding engineers through emails or phone calls to understand them thoroughly.

2.1 Data Sets

We examine misconfiguration data for one commercial system and four open-source systems. The commercial system is a storage system from COMP-A. The core software running in such system is proprietary to COMP-A. The four open-source systems include CentOS, MySQL, Apache HTTP server, and OpenLDAP. We select these software systems for two reasons: (1) they are mature and widely used, and (2) they have a large set of misconfiguration cases reported by users. While we cannot draw conclusions about any general system, our examined systems are representative of large, server-based systems. We focus only on software misconfigurations; we do not have sufficient data for hardware misconfigurations on systems running the open-source software.

COMP-A storage systems consist of multiple components including storage controllers, disk shelves, and interconnections between them (e.g., switches). These systems can be configured in a variety of ways for customers with different degrees of expertise. For instance, COMP-A offers tools that simplify system configuration. We cannot ascertain from the data whether users configured the systems directly or used tools for configuration.

The misconfiguration cases we study are from COMP-A's customer-issues database, which records problems reported by customers. For accuracy, we considered only closed cases, i.e. cases that COMP-A has provided a solution to the users. Also, to be as relevant as possible, we focused on only cases over the last two years. COMP-A's support process is rigorous, especially in comparison to open-source projects. For example, when a customer case is closed, the support engineer needs to record information about the root cause as well as resolution. Such information is very valuable for our study. There are many cases labeled as "Configuration-related" by support engineers and it is prohibitively difficult to study all of them. Therefore, we randomly sampled 1,000 cases labeled as related to configuration. Not all 1,000 cases are misconfigurations because more than half of them are simply customer questions related to how the system should be configured. Hence, we did not consider them as misconfigurations. We also pruned out a few cases for which we cannot determine whether a configuration error occurred. After careful manual examination, we identified 309 cases as misconfigurations, as shown in Table 2.

System	Total Cases	Sampled Cases	Used Cases
COMP-A	confidential	1000	309
CentOS	4338	521	60
MySQL	3340	720	55
Apache	8513	616	60
OpenLDAP	1447	472	62
Total	N/A	3329	546

Table 2: The systems we studied and the number of misconfiguration cases we identified for each of them.

Besides COMP-A storage systems, we also study four open-source systems: CentOS, MySQL, Apache HTTP server, and OpenLDAP. All of them are mature software systems, well-maintained and widely used. CentOS is an enterprise-class Linux distribution, MySQL is a database server, Apache is a web server, and OpenLDAP is a directory server.

For open-source software, the misconfiguration cases come from three sources: official user-support forums, mailing lists, and ServerFault.com (a large question-answering website focusing on system administration). Whenever necessary, scripts were used to identify cases related to systems of interest, as well as to remove those that were not confirmed by users. We then randomly sampled from all the remaining candidate cases (the candidate set sizes and the sample set sizes are also shown in Table 2) and manually examined each case to check if it is a misconfiguration. Our manual examination yielded a total of 237 misconfiguration cases from these four open-source systems. The yield ratio (used cases/sampled cases) is low for these open-source projects because we observe a higher ratio of cases that are customer questions among the samples from open source projects as compared to the commercial data.

2.2 Threats to Validity and Limitations

Many characteristic studies suffer from limitations such as the systems or workloads not being representative of the entire population, the semantics of events such as failures differing across different systems, and so on. Given that misconfiguration cases have considerably less information than ideal to work with, and that we need to perform all of the analysis manually, our study has a few more limitations. We believe that these limitations do not invalidate our results; at the same time, we urge the reader to focus on

System	Parameter	Compatibility	Component	Total
COMP-A	246 (79.6±2.4%)	31 (10.0±1.8%)	32 (10.4±1.8%)	309
CentOS	42 (70.0±3.7%)	11 (18.3±3.1%)	7 (11.7±2.6%)	60
MySQL	47 (85.5±2.3%)	0	8 (14.5±2.3%)	55
Apache	50 (83.4±2.8%)	5 (8.3±2.1%)	5 (8.3±2.1%)	60
OpenLDAP	49 (79.0±3.0%)	7 (11.2±2.3%)	6 (9.7±2.2%)	62

Table 3: The numbers of misconfigurations of each type. Their percentages and the sampling errors are also shown.

overall trends and not on precise numbers. We expect that most systems and processes for configuration errors would have similar limitations to the ones we face. Therefore, we hope that the limitations of our methodology would inspire techniques and processes that can be used to record misconfigurations more rigorously and in a format amenable to automated analysis.

Sampling: To make the time and effort manageable, we sampled the data sets. As shown in Table 2, our sample rates are statistically significant and our collections are also large enough to be statistically meaningful [10]. In our result tables, we also show the confidence interval on ratios with a 95% confidence level based on our sampling rates.

Users: The sources from which we sample contain only user-reported cases. Users may choose not to report trivial misconfigurations. Also, it is more likely that novice users may report more misconfiguration problems. We do not have sufficient data to judge whether a user is a novice or an expert. But, with new systems or major revisions of an existing system deployed to the field, there will always be new users. Therefore, our findings are still valid.

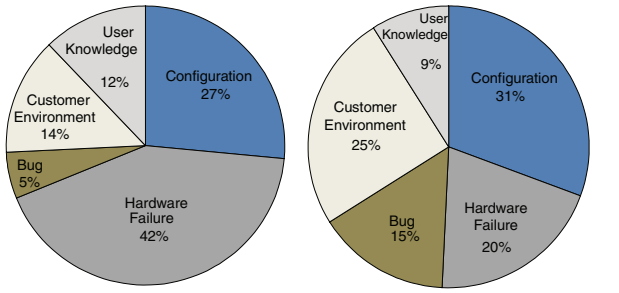
User environment: Some misconfigurations may have been prevented, or detected and resolved automatically by the system or other tools. This scenario is particularly true for COMP-A systems. At the same time, some, but not all, COMP-A customers use the tools provided by COMP-A and we cannot distinguish the two in the data.

System versions: We do not differentiate between system versions. Given that software is constantly evolving, it is possible that some of the reported configuration issues may not apply to some versions, or have already been addressed in system development (e.g., automatically correcting configuration mistakes, providing better error messages, etc.).

Overall, our study is representative of user-reported misconfigurations that are more challenging, urgent, or important.

3. IMPORTANCE OF CONFIGURATION ISSUES

We first examine how prevalent configuration issues are in the field and how severely they impact users using data from the last two years from COMP-A's customer-issues database. There are five root causes classified by COMP-A engineers after resolving each customer-reported problem: *configuration* (configuration-related), *hardware failure*, *bug*, *customer environment* (cases caused by power supplies, cooling systems, or other environmental issues), and *user knowledge* (cases where customers request information about the system). Each case is also labeled with a severity level by customer-support engineers – from "1" to "4," based on how severe the problem is in the field; cases with severity level of "1" or "2" are usually considered as high-severity cases that require prompt responses.



(a) Categorization of problem causes on all the cases

(b) Categorization of problem causes on cases with high severity

Figure 1: Root cause distribution among the customer problems reported to COMP-A

Figure 1(a) shows the distribution of customer cases based on different root causes. Figure 1(b) further shows the distribution of *high-severity* cases. We do not have the results for the open source systems due to unavailability of such labeled data (i.e., customer issues caused by hardware, software bugs, configurations, etc.).

Among all five categories, configuration-related issues contribute to 27% of the cases and are the second-most pervasive root cause of customer problems. While this number is potentially inflated by customer requests for information on configuration (as seen in our manual analysis), it shows that system configuration is nontrivial and of particular concern for customers. Furthermore, considering only high-severity cases, configuration-related issues become the most significant contributor to support cases; they contribute to 31% of high-severity cases. We expect that hardware issues are not as severe (smaller percentage of high-severity cases than of all cases) due to availability of redundancy and ease of fixes – the hardware can be replaced easily.

Finding 1.1: *Similar to the results from previous studies [11, 30, 29], data from COMP-A shows that a significant percentage (27%) of customer cases are related to configuration issues.*

Finding 1.2: *Configuration issues cause the largest percentage (31%) of high-severity support requests.*

4. MISCONFIGURATION TYPES

4.1 Distribution among Different Types

To examine misconfigurations in detail, we first look at the different types of misconfigurations that occur in the real world and their distributions. We classify the examined misconfiguration cases into three categories (as shown in Table 3). *Parameter* refers to configuration parameter mistakes; a parameter could be either an entry in a configuration file or a console command for configuring certain functionality. *Compatibility* refers to misconfigurations related to

System	Legal	Illegal					
		Format		Value			
		Lexical Mistakes	Syntax Mistakes	Typo	Value Inconsistent w/ Other Values	Value Inconsistent w/ Environment	Others
COMP-A	114(46.3±6.1%)	10(4.1±2.4%)	5(2.0±1.7%)	3(1.2±1.3%)	73 (29.7±5.6%)	32(13.0±4.1%)	9(3.7±2.3%)
CentOS	26 (61.9±13.8%)	1(2.4±4.4%)	0	2(4.8±6.0%)	6 (14.3±10.0%)	6(14.3±10.0%)	1(2.4±6.0%)
MySQL	24(51.1±12.7%)	1(2.1±3.6%)	0	0	7(14.9%±9.0%)	8(17.0%±9.5%)	7(14.9±9.0%)
Apache	27(54.0±13.3%)	3(6.0±6.3%)	3(6.0±6.3%)	1(2.0±3.7%)	7(14.0±9.3%)	5(10.0±8.0%)	4(8.0±7.3%)
OpenLDAP	23(46.9±11.5%)	7(14.3±8.0%)	11(22.4±9.6%)	0	6(12.2±7.5%)	1(2.0±3.2%)	1(2.0±3.2%)

Table 4: The distribution of different types of parameter mistakes for each application.

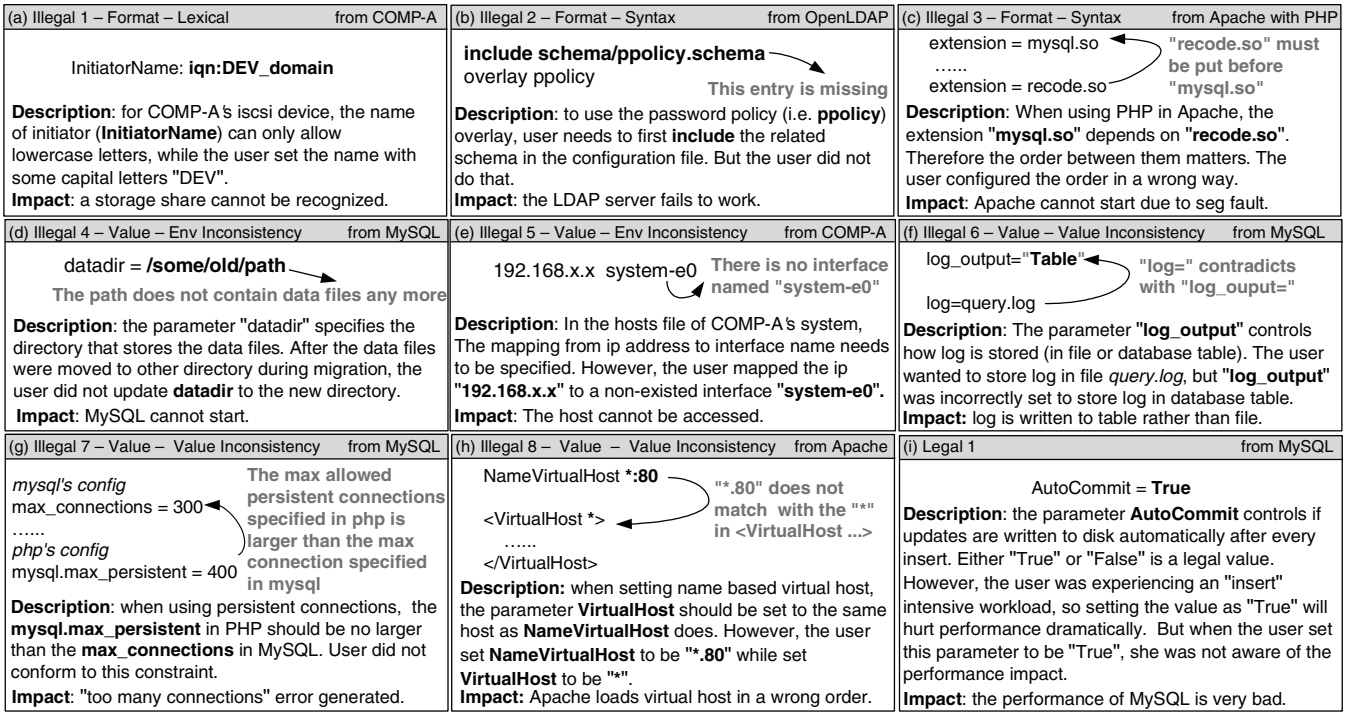


Figure 2: Examples of different types of configuration parameter related mistakes. (*legal* vs. *illegal*, *lexical error*, *syntax error* and *inconsistency error*)

software compatibility (i.e. whether different components or modules are compatible with each other). *Component* refers to other remaining software misconfigurations (e.g., a module is missing).

Finding 2.1: *Configuration parameter mistakes account for the majority (70.0%∼85.5%) of the examined misconfigurations.*

Finding 2.2: *However, a significant portion (14.5%∼30.0%) of the examined misconfigurations are caused by software compatibility and component configuration, which are not well addressed in literature.*

First, Finding 2.1 supports recent research efforts [3, 35, 38, 41] on detecting, diagnosing, or fixing parameter-based misconfigurations. Second, this finding perhaps indicates that system designers should have fewer "knobs" (i.e. parameters) for users to configure and tune. Whenever possible, auto-configuration [44] should be preferred because in many cases users may not be experienced enough to set the knobs appropriately.

While parameter-based misconfigurations are the most common, Finding 2.2 calls for attention to investigating solutions dealing with non-parameter-based configurations such as software incompatibility, etc. For example, software may need to be shipped as a complete package, deployed as an appliance (either virtual or physical), or delivered as a service (SaaS) to reduce these incompatibilities and general configuration issues.

4.2 Parameter Misconfigurations

Given the prevalence of parameter-based mistakes, we study the different types of such mistakes (as shown in Table 4), the number of parameters needed for diagnosing or fixing a parameter misconfiguration, and the problem domain of these mistakes.

Types of mistakes in parameter configuration. First, we look at parameter mistakes that clearly violate some implicit or explicit configuration rules related to format, syntax, or semantics. We call them *illegal* misconfigurations because they are unacceptable to the examined system. Figures 2(a)∼(h) show eight such examples. These types of misconfigurations may be detected automatically by checking against configuration rules.

In contrast, some other parameter mistakes are perfectly *legal*, but they are incorrect simply because they do not deliver the functionality or performance desired by users, like the example in Figure 2(i). These kinds of mistakes are difficult to detect unless users' expectation and intent can be specified separately and checked against configuration settings. More user training may reduce these kinds of mistakes, as can simplified system configuration logic, especially for things that can be auto-configured by the system.

Finding 3.1: *38.1%∼53.7% of parameter misconfigurations are caused by illegal parameters that clearly violate some format or semantic rules defined by the system, and can be potentially detected by checkers that inspect against these rules.*

Finding 3.2: *However, a large portion (46.3% ∼61.9%) of the parameter misconfigurations have perfectly legal parameters but do not deliver the functionality intended by users. These cases are more difficult to detect by automatic checkers and may require more user training or better configuration design.*

We subcategorize illegal parameter misconfigurations into *illegal format*, in which some parameters do not obey format rules such as lower case, field separators, etc.; and *illegal value*, in which the parameter format is correct but the value violates some constraints, e.g., the value of a parameter should be smaller than some threshold. We find that illegal-value misconfigurations are more common than illegal-format misconfigurations in most systems, perhaps because format is easier to test against and thereby avoid.

Illegal format misconfigurations include both *lexical* and *syntax* mistakes. Similar to lexical and syntax errors in program languages, a *lexical* mistake violates the grammar of a single parameter, like the example shown in Figure 2(a); a *syntax* mistake violates structural or order constraints of the format, like the example shown in Figure 2(b) and 2(c). As shown in Table 4, up to 14.3% of the parameter misconfigurations are lexical mistakes, and up to 22.4% are syntax mistakes.

Illegal value misconfigurations mainly consist of two type of mistakes, "*value inconsistency*" and "*environment inconsistency*". *Value inconsistency* means that some parameter settings violate some relationship constraints with some other parameters, while *environment inconsistency* means that some parameter's setting is inconsistent with the system environment (i.e., physical configuration). Figure 2(d) and 2(e) are two environment inconsistency examples. As shown in Table 4, value inconsistency accounts for 12.2%∼29.7% of the parameter misconfigurations, while environment inconsistency contributes 2.0%∼17.0%. Both can be detected by some well-designed checkers as long as the constraints are known and enforceable.

Figure 2(f), 2(g), and 2(h) present three value-inconsistency examples. In the first example, the name of the log file is specified while the log output is chosen to be database table. In the second example, two parameters from two *different* but related configuration files contradict each other. In the third example, two parameters, *NameVirtualHost* and *VirtualHost*, have unmatched values ("*.80" v.s. "*").

Finding 4: *A significant portion (12.2%∼29.7%) of parameter mistakes are due to value-based inconsistency, calling for an inconsistency checker or a better configuration design that does not require users to worry about such error-prone consistency constraints.*

Number of erroneous parameters. As some previous work on detecting or diagnosing misconfiguration focuses on only *single* configuration parameter mistakes, we look into what percentages of parameter mistakes involve only a single parameter.

System	Number of Involved Parameters		
	One	Multiple	Unknown
COMP-A	117(47.6%±6.1%)	117(47.6%±6.1%)	12(4.8%±2.6%)
CentOS	30(71.4%±12.8%)	10(23.8%±12.1%)	2(4.8%±6.0%)
MySQL	35(74.5%±11.0%)	11(23.4%±10.7%)	1(2.1%±3.6%)
Apache	31(62.0%±13.0%)	16(32.0%±12.4%)	3(6.0%±6.3%)
OpenLDAP	18(36.7%±11.1%)	30(61.2%±11.2%)	1(2.0%±3.2%)
System	**Number of Fixed Parameters**		
	One	Multiple	Unknown
COMP-A	189(76.8%±5.1%)	44(17.9%±4.7%)	13(5.3%±2.7%)
CentOS	33(78.6%±11.7%)	7(16.7%±10.6%)	2(4.8%±6.1%)
MySQL	39(83.0%±9.5%)	7(14.9%±9.0%)	1(2.1%±3.6%)
Apache	33(66.0%±12.7%)	14(28.0%±12.0%)	3(6.0%±6.3%)
OpenLDAP	29(59.2%±11.3%)	17(34.7%±11.0%)	3(6.1%±5.5%)

Table 5: The number of parameters in the configuration parameter mistakes.

Table 5 shows the number of parameters involved in configuration as well as the number of parameters that were changed to fix the misconfiguration. These numbers may not be the same because a mistake may involve two parameters, but can be fixed by changing only one parameter. Our analysis indicates that about 23.4%∼61.2% of the parameter mistakes involve multiple parameters. Examples of cases where multiple parameters are involved are cases with value inconsistencies (see above).

In comparison, about 14.9%∼34.7% of the examined misconfigurations require fixing multiple parameters. For example, the performance of a system could be influenced by several parameters. To achieve the expected level of performance, all these parameters need to be considered and set correctly.

Finding 5.1: *The majority (36.7%∼74.5%) of parameter mistakes can be diagnosed by considering only one parameter, and an even higher percentage(59.2%∼83.0%) of them can be fixed by changing the value of only one parameter.*

Finding 5.2: *However, a significant portion (23.4%∼61.2%) of parameter mistakes involve more than one parameter, and 14.9%∼34.7% require fixing more than one parameter.*

Problem domains of parameter mistakes. We also study what problem domains each parameter mistake falls under. We decide the domain based on the functionality of the involved parameter. Four major problem domains – network, permission/privilege, performance, and devices – are observed. Overall, 18.3% of examined parameter mistakes relate to how the network is configured; 16.8% relate to permission/privilege; 7.1% relate to performance adjustment. For the COMP-A systems and CentOs (the OSes), 8.5%∼26.2% of examined parameter mistakes are about device configurations.

4.3 Software Incompatibility

Besides parameter-related mistakes, software incompatibility is another major cause of misconfigurations (up to 18.3%, see Table 3). Software-incompatibility issues refer to improper combinations of components or their versions. They could be caused by incompatible libraries, applications, or even operating system kernels.

One may think that system upgrades are more likely to cause software-incompatibility issues, but we find that only 18.5% of the software-incompatibility issues are caused by upgrades. One possible reason is that both developers and users already put significant effort into the process of upgrades. For example, COMP-A provides a tool to help with upgrades that creates an easy-to-understand report of all known compatibility issues, and recommends ways to resolve them.

Some of the misconfiguration cases we analyze show that package-management systems (e.g., RPM [34] and Debian dpkg [8]) can help address many software-incompatibility issues. For example, in one of the studied cases, the user failed to install the *mod_proxy_html* module because the existing *libxml2* library was not compatible with this module.

Package-management systems may work well for systems with a standard set of packages. For systems that require multiple applications from different vendors to work together, it is more challenging. An alternative to package management systems is to use self-contained packaging, i.e. integrating dependent components into one installation package and minimizing the requirements on the target system. To further reduce dependencies, one could deliver a system as virtual machine images (e.g., Amazon Machine Image) or appliances (e.g., COMP-A's storage systems). The latter may even eliminate hardware-compatibility issues.

4.4 Component Misconfiguration

Subtype	Number of Cases
Missing component	15(25.9%)
Placement	13(22.4%)
File format	3(5.2%)
Insufficient resource	15(25.7%)
Stale data	3(5.2%)
Others	9(15.5%)

Table 6: Subtypes of component misconfigurations.

Component misconfigurations are configuration errors that are neither parameter mistakes nor compatibility problems. They are more related to how the system is organized and how resources are supplied. A sizable portion (8.3%∼14.5%) of our examined misconfigurations are of this category. Here, we further classify them into the following five subtypes based on root causes: (1) *Missing component*: certain components (modules or libraries) are missing; (2) *Placement*: certain files or components are not in the place expected by the system; (3) *File format*: the format of a certain file is not acceptable to the system. For example, an Apache web server on a Linux host cannot load a configuration file because it is in the MS-DOS format with unrecognized new line characters. (4) *Insufficient resource*: the available resources are not enough to support the system functionality (e.g., not enough disk space); (5) *Stale data*: stale data in the system prevents the new configuration. Table 6 shows the distribution of the subtypes of component misconfigurations. Missing components, placement issues, and insufficient resources are equally prominent.

4.5 Mistake Location

Table 7 shows the distribution of configuration error locations. Naturally, most misconfigurations are contained in the target application itself. However, many misconfigurations also span to places beyond the application. The administrators also need to consider other parts of the system, including file-system permissions/capacities, operating-system modules, other applications running in the system, network configuration, etc. So looking at only the application itself is not enough to diagnose and fix many configuration errors.

Finding 6: *Although most misconfigurations are located within each examined application, still a significant portion (21.7%∼57.3%) of cases involve configurations beyond the application itself or span across multiple hosts.*

5. SYSTEM REACTION TO MISCONFIGURATIONS

In this section, we examine system reactions to misconfigurations, focusing on whether the system detects the misconfiguration and on the error messages issued by the system.

5.1 Do Systems Detect and Report Configuration Errors?

Proactive detection and informative reporting can help diagnose misconfigurations more easily. Therefore, we wish to understand whether systems detect and report configuration errors. We divide the examined cases into three categories based on how well the system handles configuration errors (Table 8). Cases where the systems and associated tools detect, report, recover from (or help the user correct) misconfigurations may not be reported by users. Therefore, the results in this section may be especially skewed by the available data. Nevertheless, there are interesting findings that arise from this analysis.

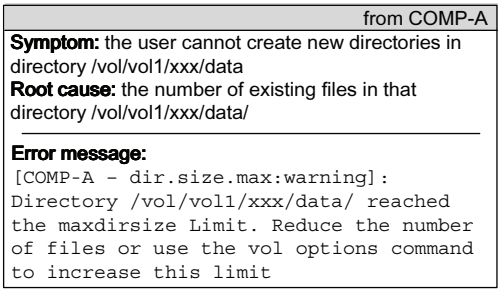


Figure 3: A misconfiguration case where the error message pinpoints the root cause and tells the user how to fix it.

We classify system reactions into *pinpoint reaction*, *indeterminate reaction*, and *quiet failure*.

A pinpoint reaction is one of the best system reactions to misconfigurations. The system not only detects a configuration error but also pinpoints the exact root cause in the error

System	Inside	FS	OS-Module	Network	Other App	Environment	Others
COMP-A	132(42.7±3.0%)	23(7.4±1.6%)	3(1.0±0.6%)	53(17.2±2.3%)	82(26.5±2.7%)	5(1.6±0.8%)	11(3.6±1.1%)
CentOS	26(43.3±4.0%)	2(3.3±1.4%)	12(20.0±3.2%)	4(6.7±2.0%)	11(18.3±3.1%)	2(3.3±1.4%)	3(5.0±1.8%)
MySQL	27(49.1±3.2%)	10(18.2±2.5%)	6(10.9±2.0%)	1(1.8±0.9%)	6(10.9±2.0%)	4(7.3±1.7%)	1(1.8±0.9%)
Apache	47(78.3±3.1%)	3(5.0±1.7%)	3(5.0±1.7%)	3(5.0±1.7%)	3(5.0±1.7%)	0	1(1.7±1.0%)
OpenLDAP	39(62.9±3.4%)	2(3.2±1.3%)	1(1.6±0.9%)	0	17(27.4±3.3%)	1(1.6±0.9%)	2(3.2±1.3%)

Table 7: The location of errors. "Inside": inside the target application. "FS": in file system. "OS-Module": in some OS modules like SELinux. "Network": in network settings. "Other App": in other applications. "Environment": other environment like DNS service.

System	Pinpoint Reaction	Indeterminate Reaction	Quiet Failure	Unknown
COMP-A	48(15.5±2.2%)	153(49.5±3.0%)	74(23.9±2.6%)	34(11.0±1.9%)
CentOS	7(11.7±2.4%)	33(55.0±3.7%)	16(26.7±3.3%)	4(6.7±1.9%)
MySQL	4(7.2±1.7%)	26(47.3±3.2%)	13(23.6±2.8%)	12(21.8±2.7%)
Apache	8(13.3±2.6%)	28(46.7±3.8%)	16(26.7±3.4%)	8(13.3±2.6%)
OpenLDAP	9(14.5±2.6%)	28(45.2%±3.7%)	14(22.6±3.1%)	11(17.7±2.8%)

(a)

System	Mysterious Symptoms w/o Message
COMP-A	26(8.4±1.7%)
CentOS	4(6.7±1.9%)
MySQL	9(16.4±2.4%)
Apache	3(5.0±1.7%)
OpenLDAP	3(4.8±1.5%)

(b)

Table 8: How do systems react to misconfigurations? Table (a) presents the number of cases in each category of system reaction. Table (b) presents the number of cases that cause mysterious crashes, hangs, etc. but do not provide any messages.

message (see a COMP-A example in Figure 3). As shown in Table 8 (a), more than 85% of the cases do *not* belong to this category, indicating that systems may *not* react in a user-friendly way to misconfigurations. As previously discussed, the study includes only reported cases. Therefore, some misconfigurations with good error messages may have already been solved by users themselves and thus not reported. So in reality, the percentage of pinpoint reaction to misconfiguration may be higher. However, considering the total number of misconfigurations in the sources we selected is very large, there are still a significant number of misconfigurations for which the examined systems do not pinpoint the misconfigurations.

An indeterminate reaction is a reaction that a system does provide some information about the failure symptoms (i.e., manifestation of the misconfiguration), but does not pinpoint the root cause or guide the user on how to fix the problem. 45.2%∼55.0% of our studied cases belong to this category.

A quiet failure refers to cases where the system does not function properly, and it further does not provide any information regarding the failure or the root cause. 22.6%∼26.7% of the cases belong to this category. Diagnosing them is very difficult.

Finding 7: *Only 7.2%∼15.5% of the studied misconfiguration problems provide explicit messages that pinpoint the configuration error.*

Quiet failures can be even worse when the misconfiguration causes the system to misbehave in a mysterious way (crash, hang, etc.) just like software bugs. We find that such behavior occurred in 5%∼8% of the cases (Table 8 (b)).

Why would misconfigurations cause a system to crash or hang unexpectedly? The reason is intuitive: since configuration parameters can also be considered as a form of input, if a system does not perform validity checking and prepare for illegal configurations, it may lead to system misbehavior. We describe two such scenarios below.

Crash example: A web application used both *mod_python* and *mod_wsgi* modules in an Apache httpd server. These two modules used two different versions of Python, which caused segmentation fault errors when trying to access the web page.

Hang example: A server was configured to authenticate via LDAP with the *hard* bind policy, which made it keep connecting to the LDAP server until it succeeded. However, the LDAP server was not working, so the server hung when the user added new accounts.

Such misbehavior is very challenging to diagnose because users and support engineers may suspect these unexpected failures to have been caused by a bug in the system instead of a configuration issue (of course, one may argue that, in a way it can also be considered to be a bug). If the system is built to perform more thorough configuration validity-checking and avoid misconfiguration-caused misbehavior, both the cost of support and the diagnosis time can be reduced.

Finding 8: *Some misconfigurations have caused the systems to crash, hang, or have severe performance degradation, making failure diagnosis a challenging task.*

We further study if there is a correlation between the type of misconfiguration and the difficulty for systems to react. We find that it is more difficult to have an appropriate reaction for software-incompatibility issues. Only 9.3% of all the incompatibility issues have pinpoint reaction, while the same ratio for parameter mistakes and component misconfigurations is 14.3% and 15.5% respectively. This result is reasonable since global knowledge (e.g., the configuration of different applications) is often required to decide if there are incompatibility issues.

5.2 System Reaction to Illegal Parameters

Cases with illegal configuration parameters (defined in Section 4.2) are usually easier to be checked and pinpointed automatically. For example, Figure 4 is a patch from MySQL that prints a warning message when the user sets illegal (inconsistent) parameters.

System	Pinpoint Reaction	Indeterminate Reaction	Quiet Failure	Unknown
COMP-A	25(18.9%)	57(43.2%)	27(20.5%)	23(17.4%)
CentOS	4(25.0%)	7(43.8%)	5(31.3%)	0
MySQL	1(4.3%)	13(56.5%)	3(13.0%)	6(26.1%)
Apache	5(21.7%)	9(39.1%)	4(17.4%)	5(21.7%)
OpenLDAP	7(26.9%)	11(42.3%)	4(15.4%)	4(15.4%)

Table 9: How do systems react to illegal parameters? The reaction category is the same as in Table 8 (a).

From MySQL mysqld.cc

```
+if (opt_logname && !(log_output_options & LOG_FILE)
+ && !(log_output_options & LOG_NONE))
+  sql_print_warning("Although a path was specified
+  for the --log option, log tables are used. To enable
+  logging to files use the --log-output option.");
```

Figure 4: A patch from MySQL that adds an explicit warning message when an illegal configuration is detected. If parameter *log_output* (value stored in variable `log_output_options`) is set as neither "FILE" (i.e. output logs to files) nor "NONE" (i.e. not output logs) but parameter *log* (value stored in variable `opt_logname`) is specified with the name of a log file, a warning will be issued because these two parameters contradict each other.

Unfortunately, systems do not detect and pinpoint a majority of these configuration mistakes, as shown in Table 9.

Finding 9: *Among 220 cases with illegal parameters that could be easily detected and fixed, only 4.3%∼26.9% of them provide explicit messages. Up to 31.3% of them do not provide any message at all, unnecessarily complicating the diagnosis process.*

5.3 Impact of Messages on Diagnosis Time

Do good error messages help engineers diagnose misconfiguration problems more efficiently? To answer this question, we calculate the diagnosis time, in hours, from the time when a misconfiguration problem was posted to the time when the correct answer was provided.

System	Explicit Message	Ambiguous Message	No Message
COMP-A	1x	13x	14.5x
CentOS	1x	3x	5.5x
MySQL	1x	3.4x	1.2x
Apache	1x	10x	3x
OpenLDAP	1x	5.3x	2.5x

Table 10: The *median* of diagnosis time for cases with and without messages (time is normalized for confidentiality reasons). *Explicit message* means that the error message directly pinpoints the location of the misconfiguration. The median diagnosis time of the cases with explicit messages is used as base. *Ambiguous message* means there are messages, but they do not directly identify the misconfiguration. *No message* is for cases where no messages are provided.

Table 10 shows that the misconfiguration cases with explicit messages are diagnosed much faster. Otherwise, engineers have to spend much more time on diagnosis, where the median of the diagnosis time is up to 14.5 times longer.

Finding 10: *Messages that pinpoint configuration errors can shorten the diagnosis time 3 to 13 times as compared to the cases with ambiguous messages or 1.2 to 14.5 times as compared to the cases with no messages.*

To improve error reporting, two types of approaches can be adopted. A white-box approach [43] uses program analysis to identify the state that should be captured at each logging statement in source code to minimize ambiguity in error messages. When source code is not available, a black-box approach, such as Clarify [12], can be taken instead. Clarify associates the program's runtime profile with ambiguous error report, which enables improved error reporting.

Interestingly, for some of the systems (Apache, MySQL, and OpenLDAP), engineers seem to spend more time (2∼4 times longer) diagnosing cases with ambiguous messages than cases with no messages at all. There are several potential reasons. First, incorrect or irrelevant messages can sometimes mislead engineers, directing them down a wrong path. Figure 5 shows such an example. Based on the message provided by the client, both the support engineers and the customers thought the problem was on the client end, so they made several attempts to set certificates, but the root cause turned out to be a problem in the configuration on the server side. This indicates that the accuracy of messages is critical to the diagnosis process. Providing misleading messages may be worse than providing nothing at all.

from COMP-A

Symptom: When the user tried to connect the admin web site, the web browser (Firefox) threw a *misleading* error message asking for new certificate.

Root cause: The "*httpd.admin.ssl.enable*" parameter was set to be "*on*" in a COMP-A server.

Error message:

```
You have received an invalid certificate.
Please contact the administrator and get
a new certificate containing a unique
serial number.
(error code: sec_error_reused_issuer)
```

Figure 5: A misconfiguration case where the error message misled the customer and the support engineers.

Second, in some cases, symptoms and configuration-file content are already sufficient for support engineers or experts to resolve the problem. For these cases, whether there are error messages is less important. For example, many cases from MySQL related to performance degradation do not have error messages, but it was relatively easy for experts to solve those problems by looking only at the configuration file. However, even for these cases, if the system could give good-quality messages, users may be able to solve these problems themselves.

Finding 11: *Giving an irrelevant message may be worse than not giving message at all for diagnosing misconfiguration. Some irrelevant messages could mislead users to chase down the wrong path. In three of the five studied systems, statistical data shows that ambiguous messages may lead to longer diagnosis time compared to not having any message.*

We further performed a preliminary study on what kind of error messages are more useful in reducing diagnosis time. Specifically, we read through the misconfiguration cases that have explicit messages and are parameter mistakes (a total of 62 cases). Besides that all these cases pinpoint the root cause of the failure (which is our definition of *explicit*), 69.4% of them further mention the parameter name in the message; 6.5% of even further point out the parameter's location within the configuration file. However, we do not find strong correlation between the diagnosis time and this extra information (e.g., parameter name) in the explicit messages. A more comprehensive study on this topic is a good avenue for future work.

6. CAUSES OF MISCONFIGURATIONS

6.1 When Do Misconfigurations Happen?

There are many ways to look at the reasons that cause a misconfiguration. Here, we examine only a couple. First, when a misconfiguration happens, i.e. whether it happens at the user's first attempt to access certain functionality, or the system used to work but does not work any more due to various changes. Based on this, we categorize the misconfiguration cases into two categories (Table 11): (1) *Used-to-work* and (2) *First-time use*.

System	Used-to-Work	First-Time Use	Unknown
COMP-A	100(32.4±2.8%)	165(53.4±3.0%)	44(14.2±2.1%)
CentOS	10(16.7±3.0%)	40(66.6±3.8%)	10(16.7±3.0%)
MySQL	3(5.5±1.5%)	45(81.8±2.5%)	7(12.7±2.2%)
Apache	2(3.3±1.4%)	40(66.7±3.6%)	18(30.0±3.5%)
OpenLDAP	2(3.2±1.3%)	57(91.9±1.6%)	3(4.8±1.6%)

Table 11: The number of misconfigurations categorized by *used-to-work* and *first-time use*.

One may think that most misconfigurations happen when users configure a system for the first time. As our results show, it is indeed the case, especially for relatively simple systems (MySQL, Apache, and OpenLDAP). The causes for the misconfigurations during first-time use can be the inadequate knowledge of personnel, flawed design of the system, or even inconsistent user manuals [33].

However, for more complex systems, COMP-A and CentOS, a significant portion (16.7%∼32.4%) of the misconfigurations happen in the middle of the system's lifetime. There could be two major reasons. First, these systems have more frequent changes (upgrades, reconfiguration, etc.) in their lifetime. Second, the configuration is more complicated, so it takes a long time for users to master.

Finding 12: *The majority of misconfigurations are related to first-time use of desired functionality. For more complex systems, a significant percentage (16.7%∼32.4%) of misconfigurations were introduced into systems that used to work.*

6.2 Why Do Systems Stop Working?

To further examine the causes of used-to-work cases, we categorize the 100 cases of this category from COMP-A based on their root causes (Figure 6).

Collateral damage refers to cases when users made configuration changes for some new functionality but accidentally broke existing functionality. It accounts for 29.0% of the used-to-work cases from COMP-A. To avoid such collateral damages, it might be useful if users can be warned by the configuration management/change tool about the side-effects of their changes.

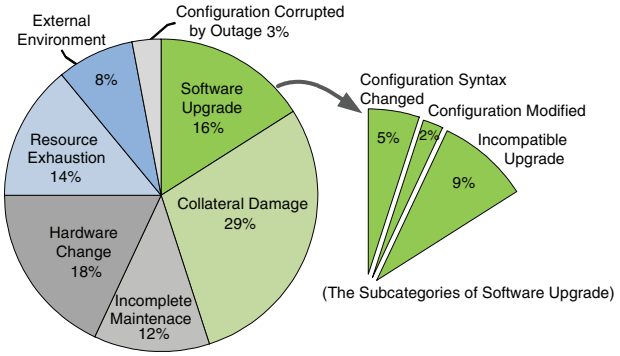


Figure 6: The cause distribution for the *used-to-work* misconfigurations at COMP-A (we also subcategorize the cases caused by *software upgrade*).

Incomplete maintenance refers to cases when some regular maintenance tasks introduced incomplete configuration changes. 12.0% of the used-to-work cases from COMP-A belong to this category. For example, when an administrator does a routine periodic password change to certain accounts but forgets to propagate it to all affected systems, some systems would not be able to authenticate these accounts.

In addition, configuration could also be *corrupted by outage* (3.0%) or be modified accidentally by some (2.0%) *software upgrades* (Figure 6). To sum up, 46% of the examined used-to-work misconfiguration cases from COMP-A are caused by configuration-parameter changes due to various reasons, including configuring other features, routine maintenance, system outages, or software upgrades. To diagnose and fix these cases, it is useful for systems to automatically keep track of configuration changes [24], and even better, help users to pinpoint which change is the culprit [41].

Another major cause is *hardware change* (18.0%). When customers upgrade, replace or reorganize hardware (e.g., moving a disk from one server to another), it can cause problems if they forget to change related configuration parameters accordingly.

Resource exhaustion (14.0%) can also affect a previously working system. For example, in one of the studied cases, a database system hung and did not work properly even after rebooting because the data disks became full.

Finally, *external environment* changes could also be harmful to previously working systems. They account for 8.0% of used-to-work cases from COMP-A. For example, in one of the studied cases, a system suffered from severe performance degradation because its primary DNS server went offline accidentally. Such changes are error prone and problematic, because different systems may be managed by different administrators who may not communicate with each other in a timely manner about their changes.

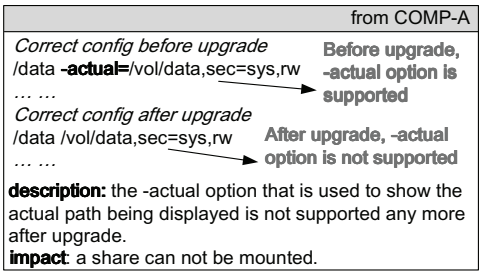


Figure 7: A misconfiguration example where the syntax of configuration files has changed after upgrade. A previously working NFS mounting configuration is no longer valid, because the option *actual* became deprecated after upgrade.

Software upgrades, as one may expect, is another major cause of misconfigurations that break a previously working system. It accounts for 16% of the "used-to-work" cases from COMP-A. We further subcategorize it into three types. First, a new software release may have changed the configuration file syntax or format requirements, making the old configuration file invalid. Figure 7 gives such an example. Second, some automatic upgrade processes may silently modify certain configuration parameters (e.g. set them to default values) without users' awareness. Third, software upgrades may cause incompatibilities among components.

In order to prevent misconfigurations caused by software upgrades, systems should provide automatic upgrade tools or at least detailed upgrade instructions [24, 7]. The upgrade process should also take users' existing configurations into consideration.

Finding 13: *By looking into the 100 used-to-work cases (32.4% of the total) at COMP-A, 46% of them are attributed to configuration parameter changes due to routine maintenance, configuring for new functionality, system outages, etc, and can benefit from tracking configuration changes. The remainder are caused by non-parameter related issues such as hardware changes (18%), external environmental changes (8%), resource exhaustion (14%), and software upgrades(14%).*

7. IMPACT OF MISCONFIGURATIONS

We analyzed the severity of customer-reported issues from COMP-A (Section 3) and found that a large percentage (31%) of high-impact issues were related to system configuration. In this section, we analyze the severity of the specific misconfiguration cases used in our study, particularly from the viewpoint of system availability and performance. We divide the misconfiguration cases into three categories, as shown in Table 12: (1) the system becomes *fully unavailable*; (2) the system becomes *partially unavailable*, i.e. it cannot deliver certain desired features; and (3) the system suffers from severe *performance degradation*. We do expect the results to be skewed towards the more severe, causing users to report them as issues more than simpler cases.

We find 9.7%∼27.3% of the misconfigurations cause the system to become fully unavailable. This shows again that misconfigurations can be a severe threat to system availability.

System	Fully Unavailable	Partially Unavailable	Performance Degradation
COMP-A	41 (13.3±2.1%)	247 (79.9±2.4%)	21 (6.8±1.5%)
CentOS	12 (20.0±3.2%)	47 (78.3±3.3%)	1 (1.7±1.0%)
MySQL	15 (27.3±2.9%)	29 (52.7±3.2%)	11 (20.0±2.6%)
Apache	15 (25.0±3.3%)	44 (73.3±3.4%)	1 (1.7±1.0%)
OpenLDAP	6 (9.7±2.2%)	52 (83.9±2.7%)	4 (6.4±1.8%)

Table 12: The impact distribution of the misconfiguration cases from all the studied systems.

Moreover, up to 20.0% of the misconfigurations cause severe performance degradation, especially for systems such as database servers that are performance-sensitive and require some nontrivial tuning based on users' particular workloads, infrastructure, and data sizes. For example, the official performance tuning guides for MySQL and Oracle have more than 400 pages, and mention tens, even hundreds of configuration parameters that are related to performance. The percentage of misconfigurations causing performance issues here might be an underestimate of performance problems in the field, since some trivial performance issues introduced by misconfigurations may not be reported by the user.

Finding 14: *Although most studied misconfiguration cases only lead to partial unavailability of the system, 16.1%∼47.3% of them make the systems fully unavailable or cause severe performance degradation.*

The next question is whether different types of misconfigurations have different impact characteristics. Therefore, we also examine the impact of each type of misconfiguration; the results are shown in Table 13.

Misconfig Type	Fully Unavailable	Partially Unavailable	Performance Degradation
Parameters	59 (**13.6%**)	342 (78.8%)	33 (7.6%)
Compatibility	14 (**25.9%**)	38 (70.4%)	2 (3.7%)
Component	16 (**27.6%**)	39 (67.2%)	3 (5.2%)

Table 13: The impact on different types of misconfiguration cases. The data is aggregated for all the examined systems. The percentage shows the ratio of a specific type of misconfiguration (e.g., parameter mistake) that leads to a specific impact level (e.g., full unavailability).

We find that, compared to configuration parameter mistakes, software compatibility and component configuration errors are more likely to cause full unavailability of the system. 25.9% of the software compatibility issues and 27.6% of the component configuration errors make systems fully unavailable, whereas this ratio is only 13.6% for parameter-related misconfigurations.

The above results are not surprising because what components are used and whether they are compatible can easily prevent systems from even being able to start. In contrast, configuration-parameter mistakes, especially if the parameter is only for certain functionality, tend to have a much more localized impact.

In addition to having a more severe impact, compatibility and component configuration mistakes can be more difficult to fix. They usually require greater expertise from users. For example, in one of the misconfiguration cases of CentOS, the user could not mount a newly created ReiserFS file system, because the kernel support for this ReiserFS file system was

missing. The user needed to install a set of libraries and kernel modules and also modify configuration parameters in several places to get it to work.

8. RELATED WORK

Characteristic studies on operator errors: Several previous studies have examined the contribution of operation errors or administrator mistakes [11, 22, 23, 27, 29, 30]. For example, Jim Gray found that 42% of system failures are due to administration errors [11]. Patterson et al. [30] also observed a similar trend in telephone networks. Murphy et al. [22] found that the percentage of failures due to system management is increasing over time. Oppenheimer et al. [29] studied the failures of the Internet services and found that configuration errors are the largest category of operator errors. Nagaraja et al. [23] also had similar findings from a user study.

To the best of our knowledge, very few studies have analyzed misconfigurations in detail and examined the subtypes, root causes, impacts and system reactions to misconfigurations, especially in both *commercial* and open source systems with a large set of real-world misconfigurations.

Detection of misconfigurations: A series of efforts [9, 21, 24, 38, 40] in recent years have focused on detecting misconfigurations. The techniques used in PeerPressure [38] and its predecessor Strider [40] have been discussed in the Introduction. Microsoft Baseline Security Analyzer (MBSA) [21] detects common *security-related* misconfigurations by checking configuration files against predefined rules; security is one of the important impact categories we have not focused on in our study. NetApp's AutoSupport-based health management system [24] checks the validity of configurations against "golden templates", focusing on compatibility and component issues (which are likelier to cause full availability according to our study).

Diagnosis of misconfigurations: Besides detection, another series of research efforts [41, 35, 2, 3] focus on diagnosing problems after the errors happen. We have already discussed AutoBash [35], ConfAid [3], and Chronus [41] in the Introduction. The applicability of Chronus depends on how many misconfigurations belong to the "used-to-work" category; according to our study, it is a significant percentage for more complex systems. A follow-up work to AutoBash by Attariyan et al. [2] leverages system call information to track the causality relation, which overcomes the limitations of the Hamming distance comparison used in AutoBash to further enhance accuracy. Similar to [2], Yuan et al. [42] use machine learning techniques to correlate system call information to problem causes in order to diagnose configuration errors. Most of these works focused on parameter-related misconfigurations.

Tolerance of misconfigurations: Some research work [35, 4] can help fix or tolerate misconfigurations. In addition to AutoBash [35], Undo [4] uses checkpoints to allow administrators to have a chance to roll back if they made some misconfigurations. Obviously, it assumes that the system used to work fine, thus addressing a significant number of cases for more complex systems.

Avoidance of misconfigurations: One approach to avoid misconfiguration is to develop tools to configure the system automatically. SmartFrog [1] uses a declarative language to describe software components and configuration parameters, and how they should connect to each other. Configurations can then be automatically generated to greatly mitigate human errors. Similarly, Zheng et al. [44] leverage custom-specified templates to automatically generate the correct configuration for a system. Kardo [18] adopts machine learning techniques to automatically extract the solution operations out of the user's UI sequence and apply them automatically. The significant percentage of "illegal configuration parameters" provides some supporting evidence and also shows the benefits of the above approaches.

A more fundamental approach is to design better configuration logic/interface to avoid misconfigurations. Maxion et al. [20] discovered that many misconfigurations for NTFS permissions are due to the configuration interfaces not providing adequate information to users. Therefore, they proposed a new design of the interface with subgoal support that can effectively reduce the configuration errors on NTFS permissions by 94%.

Misconfiguration injection: As mentioned in the Introduction, a misconfiguration-injection framework like ConfErr [17] is very useful for evaluating techniques for detecting, diagnosing, and fixing misconfigurations. Our study can be beneficial for such framework to construct a more accurate misconfiguration model.

Online validation: Another avenue of work [7, 23, 27, 28] focus on validating the system for detecting operator mistakes. Nagaraja et al. [23] developed a validation framework which can detect operator mistakes before deployment by comparing against the comparator functions provided by users. A follow-up work by Oliveira et al. [27] validates database system administrations. Another follow-up work by Oliveira et al. [28] addresses the limitation of the previous validation system, which does not protect against human errors directly performed on the production system. Mirage [7] also has a subsystem for validating system upgrades.

Miscellaneous: Wang et al. [39] used reverse engineering to extract the security-related configuration parameters automatically. Users can leverage the approach to slice the configuration file and see if the security-related parameters are correct. Ramachandran et al. [32] extracted the correlations between parameters, which can be used to detect some of the inconsistent-parameter misconfigurations in our study. Rabkin et al. [31] found that the configuration space after canonicalization is not very big after having analyzed seven open source applications. Therefore a thorough test of different configuration parameters might be possible for certain applications if input is generated in a smart way.

As we discussed in the Introduction, our characteristic study of real-world misconfigurations would be useful in providing some guidelines to evaluate, improve, and extend some of the above work on detecting, diagnosing, fixing, and injecting misconfigurations.

9. CONCLUSIONS

System configuration lies in the gray zone between the developers of a system and its users. The responsibility for creating correct configurations lies with both parties; the developer should create intuitive configuration logic, build logic that detects errors, and convey configuration knowledge to users effectively; the user should imbibe the knowledge and manage cross-application or cross-vendor configurations. This shared responsibility is non-trivial to efficiently achieve. For example, there is no obviously "correct" way to build configuration logic; also, unlike fixing a bug once, every user of the system has to be educated on the right way to configure the system. Perhaps as a result, misconfigurations have been one of the dominant causes of system issues and is likely to continue so.

We have performed a comprehensive characteristic study on 546 randomly-sampled real-world misconfiguration cases from both a commercial system that is deployed to thousands of customers, and four widely used open-source systems, namely CentOS, MySQL, Apache, and OpenLDAP. Our study covers several dimensions of misconfigurations, including types, causes, impact, and system reactions. We hope that our study helps extend and improve tools that inject, detect, diagnose, or fix misconfigurations. Further, we hope that the study provides system architects, developers, and testers insights into configuration-logic design and testing, and also encourages support personnel to record field configuration problems more rigorously so that vendors can learn from historical mistakes.

10. ACKNOWLEDGMENTS

We would like to express our great appreciation to our shepherd, Emmett Witchel, who was very responsive and provided us with valuable suggestions to improve our work. We also thank the anonymous reviewers for their insightful comments and suggestions. Moreover, we thank Kiran Srinivasan, Puneet Anand, Scott Leaver, Karl Danz, and James Ayscue for their feedback and insights, and thank the support engineers and developers of COMP-A storage systems and the open-source systems used in the study for their help. Finally, we greatly appreciate Soyeon Park, Weiwei Xiong, Jiaqi Zhang, Xiaoming Tang, Peng Huang, Yang Liu, Michael Lee, Ding Yuan, Zhuoer Wang, and Alexander Rasmussen for helping us proofread our work. This research is supported by NSF CNS-0720743 grant, NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS- 0347854 (career award), NSF CSR Small 1017784 grant, and a NetApp Faculty Fellowship.

11. REFERENCES

[1] P. Anderson, P. Goldsack, and J. Paterson. SmartFrog meets LCFG Autonomous Reconfiguration with Central Policy Control. In *LISA*, August 2003.

[2] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *USENIX*, June 2008.

[3] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, October 2010.

[4] A. B. Brown and D. A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *USENIX*, June 2003.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP'01*.

[6] CircleID. Misconfiguration brings down entire .se domain in sweden. `www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden/`.

[7] O. Crameri, N. Knezević, D. Kostić, R. Bianchini, and W. Zwaenepoel. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In *SOSP'07*, October 2007.

[8] Debian. The Debian GNU/Linux FAQ, Chapter 8: The Debian Package Management Tools. `http://www.debian.org/doc/FAQ/ch-pkgtools.en.html`.

[9] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, May 2005.

[10] D. Freedman, R. Pisani, and R. Purves. *Statistics, 3rd Edition*. W. W. Norton & Company., 1997.

[11] J. Gray. Why do computers stop and what can be done about it? In *Symp. on Reliability in Distributed Software and Database Systems*, 1986.

[12] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved Error Reporting for Software that Uses Black-Box Components. In *PLDI*, 2007.

[13] Hewlett-Packard. HP Storage Essentials SRM Software Suite. `http://h18000.www1.hp.com/products/quickspecs/12191_na/12191_na.pdf`.

[14] IBM Corp. IBM Tivoli Software. `http://www-01.ibm.com/software/tivoli/`.

[15] R. Johnson. More details on today's outage. `http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919`.

[16] A. Kappor. Web-to-host: Reducing total cost of ownership. In *Technical Report 200503, The Tolly Group*, May 2000.

[17] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *DSN*, June 2008.

[18] N. Kushman and D. Katabi. Enabling Configuration-Independent Automation by Non-Expert Users. In *OSDI*, October 2010.

[19] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, March 2008.

[20] R. A. Maxion and R. W. Reeder. Improving user-interface dependability through mitigation of human error. *International Journal of Human-Computer Studies*, 63, July 2005.

[21] Microsoft Corp. Microsoft Baseline Security Analyzer. 2008. `http://www.microsoft.com/technet/security/tools/MBSAHome.mspx`.

[22] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. In *Quality and Reliability Engineering International, 11(5),*, 1995.

[23] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *OSDI'04*,

October 2004.

[24] NetApp, Inc. Proactive Health Management with AutoSupport. http://media.netapp.com/documents/wp-7027.pdf.

[25] NetApp, Inc. Protection Manager. http://www.netapp.com/us/products/management-software/protection.html.

[26] NetApp, Inc. Provisioning Manager. http://www.netapp.com/us/products/management-software/provisioning.html.

[27] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Validating Database System Administration. In *USENIX'06*, 2006.

[28] F. Oliveira, A. Tjang, R. Bianchini, R. P. Martin, and T. D. Nguyen. Barricade: Defending Systems Against Operator Mistakes. In *EuroSys'10*, April 2010.

[29] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. 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)*, March 2003.

[30] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. In *Technical Report UCB//CSD-02-1175, University of California, Berkeley*, March 2002.

[31] A. Rabkin and R. Katz. Static Extraction of Program Configuration Options. In *ICSE*, May 2011.

[32] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury. Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-tiered Enterprise Applications. In *ICAC*, June 2009.

[33] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, May 2010.

[34] RPM. Rpm package manager (rpm). http://rpm.org/.

[35] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: improving configuration management with operating system causality analysis. In *SOSP*, October 2007.

[36] M. Sullivan and R. Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *FTCS*, 1991.

[37] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *International Symposium on Fault-Tolerant Computing*, 1992.

[38] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI'04*, October 2004.

[39] R. Wang, X. Wang, K. Zhang, and Z. li. Towards Automatic Reverse Engineering of Software Security Configurations. In *CCS*, October 2008.

[40] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *LISA'03*, October 2003.

[41] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI*, October 2004.

[42] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated Known Problem Diagnosis with Event Traces. In *EuroSys*, April 2006.

[43] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. In *ASPLOS*, March 2011.

[44] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic Configuration of Internet Services. In *EuroSys*, March 2007.