ConfTest: Generating Comprehensive Misconfiguration for System Reaction Ability Evaluation

Wang Li
National University of Defense Technology
P. R. China
liwang2015@nudt.edu.cn

Shanshan Li
National University of Defense Technology
P. R. China
shanshanli@nudt.edu.cn

Xiangke Liao
National University of Defense Technology
P. R. China
xkliao@nudt.edu.cn

Xiangyang Xu
National University of Defense Technology
P. R. China
xuxiangyang11@nudt.edu.cn

Shulin Zhou
National University of Defense Technology
P. R. China
zhoushulin@nudt.edu.cn

Zhouyang Jia
National University of Defense Technology
P. R. China
jiazhouyang@nudt.edu.cn

ABSTRACT

Misconfigurations are not only prevalent, but also costly on diagnosing and troubleshooting. Unlike software bugs, misconfigurations are more vulnerable to users' mistakes. Improving system reaction to misconfigurations would ease the burden of users' diagnoses. Such effort can greatly benefit from a comprehensive study of system reaction ability towards misconfigurations based on errors injection method. Unfortunately, few such studies have achieved the above goal in the past, primarily because they fail to provide rich error types or only rely on generic alternations to generate misconfigurations. In this paper, we studied 8 mature opensource and commercial software and summarized a fine-grained classification of option types. On the basis of this classification, we could extract syntactic and semantic constraints of each type to generate misconfigurations. We implemented a tool named ConfTest to conduct misconfiguration injection and further analyze system reaction abilities to various of misconfigurations. We carried out comprehensive analyses upon 4 open-source software systems. Our evaluation results show that our option classification covers over 96% of 1582 options from Httpd, Yum, PostgreSQL and MySQL.Our constraint is more fined-grained and the accuracy is more than 90% of of real constraints through manual verification. We compared the capability in finding bad system reactions between ConfTest and ConfErr, showing that the ConfTest can find nearly 3 times the bad reactions found by ConfErr.

CCS CONCEPTS

• **General and reference → Reliability**; **Evaluation**; • **Software and its engineering → Software testing and debugging**;

KEYWORDS

Misconfiguration, system reactions, constraints

EASE'17, Karlskrona, Sweden

DOI: http://dx.doi.org/10.1145/3084226.3084244

ACM Reference format:
Wang Li, Shanshan Li, Xiangke Liao, Xiangyang Xu, Shulin Zhou, and Zhouyang Jia. 2017. ConfTest: Generating Comprehensive Misconfiguration for System Reaction Ability Evaluation . In *Proceedings of EASE'17, Karlskrona, Sweden, June 15-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3084226.3084244

1 INTRODUCTION

With large software systems' growing impact on people's lives, misconfiguration has become quite a critical issue. Several researches [7, 8, 12, 13] reveal that misconfigurations are one of the major causes for deterioration of the software reliability. Yin et al. [25] report that 27% of customer cases of a commercial storage system are related to configuration errors. At the same time, widely commercial systems [18, 19, 23] suffered from misconfiguration such as outages. Besides, the seriousness of the misconfiguration problem was often underestimated, which has caused a major loss every year. According to Computer Research Association's report [3], a large number of capital outlay of IT ownership was spent during troubleshooting system misconfigurations. Unfortunately, the difficulty in diagnosis of misconfigurations is much higher than expected. This is reflected in three main aspects: (1) Root causes of misconfiguration are complicated. Figure 1(a) indicates that misconfigurations are resulted from not only human mistakes but also resulted from inappropriate software designs. (2) Misconfigurations are hard to be detected before triggered. Figure 1(b) gives an example of how software Yum failed to detect the latent configuration error. In this case, the value of option "cachedir" was incorrectly set, Yum, however, still works until it was called to access the local cache. (3) The lack of feedback also obstructs the diagnosis of misconfigurations. Figure 1(c) shows that MySQL complains "can't start server", which is helpless on diagnosing the misconfigu[illegible].

The severity of misconfigurations has inspired many research efforts on diagnosing them. Those works can be classified into program analysis approaches, statistical approaches and configuration testing approaches. Program analysis approaches [4, 6] use data flow to automatically diagnose misconfigurations. These approaches, however, have a limitation of scalability. Statistical approaches, such like PeerPressure [20], STRIDER [21] and EnCore

Misconfiguration A: symbolic_lin=0 /* MySQL */
Root cause: typo mistakes of administrator: omission 'k' in "symbolic_link"
Misconfiguration B: symbolic_link=yes
Root cause: system can't recognize the string "yes" and rollbacks the value of the parameter

(a) Complicated misconfiguration root cause

Misconfiguration: /* Yum */
cachedir=/nonexistent/directory
Symptoms:
Yum will fail if it needs to access the local cache, but works well if not

(b) Study case of Yum. Misconfigurations is latent in system until triggered by specified functions.

Misconfiguration: /* MySQL */
socket=/nonexistent/filename
Symptoms:
MySQL fails after startup and prints logs: "Can't start server : Bind on unix socket: Address already in use"

(c) Study case of MySQL. Poor feedbacks of MySQL obstruct the diagnoses of the misconfiguration.

Figure 1: Challenges in diagnosing misconfiguration

[27], diagnose misconfigurations by learning configuration rules, which requires a large collection of independent configuration settings from hundreds of machines. Configuration testing approaches like ConfErr [9] and SPEX [23] evaluate system reactions ability by generating and testing misconfigurations.

Researchers and developers would benefit greatly if there is a comprehensive study of system reaction ability of misconfiguration since it will help misconfiguration diagnosis (e.g. finding and enhancing inadequate diagnostic messages for misconfigurations). A good reaction, which means error indication and error handling, would greatly ease the burden of users diagnosing misconfigurations. Unfortunately, few such studies have been conducted in the past. Despite pioneer in configuration testing, ConfErr relies on generic alternations to generate misconfigurations, which weaken the capability of evaluating system reactions. SPEX, takes a step further, infers 5 main categories of constraints (i.e., rules that differentiate correct configurations from misconfigurations) but coarse-grained in those of option types, as a result of poor diversity, variance in reactions of misconfigurations cannot be observed by researchers and developers.

In order to harden systems against misconfiguration and improve software reliability, we explored an error-injection method to test systems and try to detect their reaction ability to misconfiguration. We evaluated system ability for misconfiguration diagnosis according to their reactions, such as failures and inadequate diagnostic messages [29]. To achieve above goal and provide a more comprehensive study of system reaction ability of misconfigurations, we implemented a tool named ConfTest.

In our work, we summarized and classified 1593 software configuration options from 8 mature open source and commercial software systems. Based on these classifications, we are able to summarize and extract fine-grained options constraints compared to previous work. Through violating these fine-grained constraints, a variety of misconfigurations are generated and injected into systems by ConfTest. We then analyzed the distribution of different system reactions and tried to reveal some design problems related to misconfiguration, based on these problems, we put forward some comments to improve software reaction ability.

The contributions of this paper are as follows:

(1) To generate more effective constraints for each type of configuration option, we summarized a comprehensive classification based on a large amount of configuration options from 8 mature open-source and commercial software. Our classification is tree based and easy for extension. Based on this classification, we proposed syntactic and semantic constraints for each type. Our evaluation results show that our option classification covers over 96% of 1582 options from Httpd, Yum, PostgreSQL and MySQL. Compared with constraints proposed by EnCore, our constraint is more finedgrained and is consistent with more than 90% of real constraints through manual verification.

(2) We implemented a tool named ConfTest to conduct misconfiguration injection and make evaluation on system reaction abilities. Based on these results, ConfTest can reveal bad reactions and design problems in the systems. We compared the capability in finding bad system reactions between ConfTest and ConfErr, the results show that the ConfTest can find nearly 3 times the bad reactions found by ConfErr.

(3) We defined 6 types for system reactions to misconfigurations. Based on these types, we calculated the distribution of system reactions and analyzed the reasons for these reactions. We found that misconfiguration of Path might be hard for system to diagnose due to lack of checking for constraints both syntactically and semantically. Our experimental results show that adequate configuration syntax checking after startup can effectively help diagnose the misconfigurations.

The remainder of this paper is organized as follows. We present constraints generation in Section 2. In Section 3, we give the process of misconfiguration generation. The evaluation is in Section 4, and Section 5 is our experience and practice. We present the related work in Section 6, and conclude our work in Section 7.

2 CONSTRAINTS GENERATION

Configuration constraints are the condition that configuration options should satisfy. For example, when administrator tries to rewrite the options in configuration files, he should follow the rules specific to options such like file path options or boolean options. To understand configuration constraints better, in this section, we studied over 1500 configuration options from several open-source and commercial software, which are widely used, and classified these options by types and generated corresbonding constraints, which can be used for misconfiguration injection. We selected Squid, Nginx, Redis, Nagios, Lighttpd (core), Puppet, SeaFile, Vsftpd in our study for their representativeness in each field. Squid is a caching proxy for web and are being increasingly used in content delivery architectures like Flickr and Wikipedia. Nginx is a light-weight web server and now used by millions of sites, including WordPress

and SourceForge. Redis is an open source in-memory data structure store, and used by Twitter, GitHub, SnapChat etc. Nagios is a commercial IT infrastructure monitoring system, with over 1 million users worldwide. Lighttpd is a web server system with rapidly redefining efficiency. Puppet is a platform for automatically delivering, operating and securing infrastructure, and used by over 30 thousands organizations. Seafile is an enterprise file sync and share platform with high reliability and performance. Currently, Seafile has over 300,000 users worldwide. Vsftpd is the default FTP server in the Ubuntu, CentOS, Fedora, NimbleX, Slackware and RHEL Linux distributions. In this section, we investigated the configuration files and related documents for each software. We manually investigated the configuration files as well as related official documents of software and recorded the detailed information of each option, such as name, default value, descriptions, etc. In this paper, we only consider options in the form of key-value pairs. To simplify our work, we convert those options of other formats into key-value pairs.

2.1 Type Taxonomy

Although previous work [16] had studies on type taxonomy, our aim is to generate option constraints for misconfiguration injection, therefore, we need a fine-grained enough classification for all configuration options. We carefully classified each option manually according to its type as we encountered it. In Table 1, we illustrate the main types of options in our work by statistics. These configuration options mainly come from software documents, such as user manuals, official guidebook etc. If this information can't be accessed from documents, we return to configuration files or even source code to speculate types the options might belong.

According to results in Table 1, we found that most options can be well presented by a few set of types. Figure 2 illustrates our classification as a tree. This classification tree can be supplemented with more software to be considered.

We evaluated our classification effectiveness on 1582 options of other four open source software systems (Httpd, MySQL, PostgreSQL and Yum) by checking whether every options can be classified into types listed Figure 2. Table 2 shows that coverage rate (option types excluding "Others") of the classification is as high as 96.5%, and at least 95.8% for each one.

While our classification can be reasonably effective, there are still some knotty problems. Some options we examined can be set as several types. For example, option "LogFile" in MySQL has default value type which is Path, at the same time, value of "LogFile" can be changed to network address. Besides, some options exist in the form of complicated structure which are hard to be converted into the form of key value pairs. Many options in Httpd are like directives with more than one argument and cannot be recognized by our method. In this case, we consider each value's type of the option. Types unclassified in Figure 2 (labeled as "Others") are also a problem, in particular, all of the classifications in our method should be taken with specific software systems and it may fail to find particular types in some situation, but our classification can be easily supplemented and new types can be easily incorporated into it.

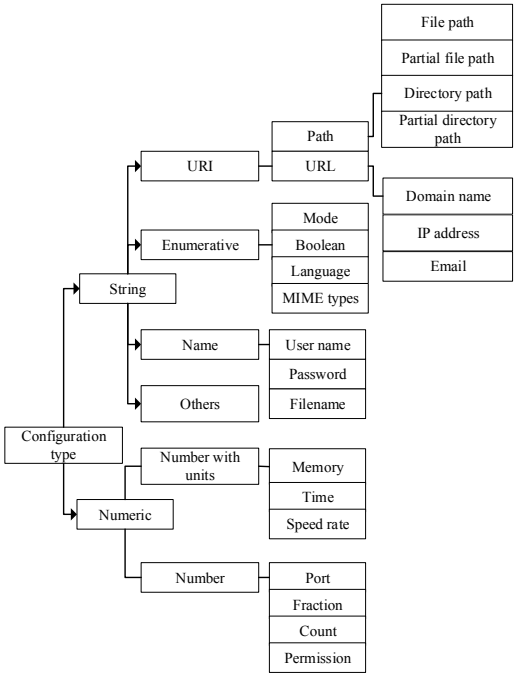


Figure 2: Type classification

2.2 Inferring Type Constraints

To evaluate system reaction ability comprehensively, we should generate varied misconfigurations as much as possible. To achieve this purpose, we try to extract fine-grained constraint of each configuration type, either by inherent constraint or from domain knowledge (e.g. RFC documents). Some previous studies tried to use program analysis to infer constraints [23], unfortunately, through large amount of manual analyses on source code of many open source software, we found there are various kinds and styles of constraint existing in source codes, it's difficult, if not impossible, to extract constraints through program analysis. We consider constraints from aspects of semantic and syntactic according to the relationship with execution environment. For example, Type PORT has a semantic constraint that it should not use a same number with a used port number because this constraint is related with other software running in its execution environment, and its value should be an integer between 0 and 65535, that is a syntactic constraint.

Similar to work in EnCore [27], our syntactic constraints can be expressed by normal forms or string patterns of their commonly used standardization. We consider all these constraints in software as syntactic constraints, and such predefined patterns are used to generate misconfigurations by violating rules of syntax to simulate human errors. Table 3 illustrates the details of the syntactic constraints. We use wildcard to represent the elements in the patterns and further explain their value range. Program analysis is also used in our inference for certain system specified constraints (e.g., Range of Count, the value set of Mode)

Table 1: Numbers of options with classification, by applications

Software	IP-address	Boolean	Mode	Path	Email	Count	Permission	Port	Domain Name	Name	Others	-
Squid	11	73	46	26	4	111	0	7	11	34	16	339
Nginx	7	114	124	65	1	186	5	0	7	104	24	637
Redis	2	13	9	6	0	33	0	1	0	0	6	70
Nagios	0	33	10	17	2	37	0	0	0	16	8	123
Lighttpd	1	12	6	5	0	12	0	0	1	14	4	55
Puppet	0	45	1	77	0	19	0	1	6	57	2	208
SeaFile	0	15	2	1	0	13	0	1	3	1	2	38
Vsftpd	2	73	2	19	0	16	0	4	2	1	2	123

Table 2: Coverage rate of classification

Software	Options	Coverage rate
Httpd	564	543(96.2%)
MySQL	671	643(95.8%)
PostgreSQL	273	270(98.9%)
Yum	74	71(95.9%)
Total	1582	1527(96.5%)

Unlike syntactic constraints, semantic constraints reflect the complex relationship between software and environments. We consider that semantic constraints consist of two parts: the semantic constraints of option's value and the environment related attributes of options. As for the former, Table 4 lists the semantic constraints for value of each configuration option type. These constraints do exist while they cannot be expressed by syntactic form. With syntactic constraints, you might get a URL with correct syntactic form, but whether it can be accessed depends on the system context, for example, whether it is forbidden by firewall. These semantic constraints are related to environment and deserve our high attention.

Each configuration option has attributes, which reflect system context. Although they don't have a direct relationship with the value of configuration option, their values must meet some requirements to keep the consistent of system. In Table 5, we augment each option type with environment attributes, which reflect system context, and each attribute is assigned with a type. We infer the semantic constraints with the environment information collected and domain knowledge. Inspired by EnCore [27], we summarized environment related configuration into 7 main sources: Network, Services, Hardware, File System, Security, and Environment Variable. For each option type, we collect the relevant execution information as listed in Table 4 and Table 5.

As illustrated in Table 6, our constraints can be consistent with over 90% of options. Compared to the predefined patterns EnCore [27] uses to infer certain option types, our constraints are more fine-grained and flexible, because we consider not only the string pattern but also the data range (e.g. the value set of Mode, the valid range of Count). SPEX can also infer such constraints but not fine-grained enough in syntactic constraints, and incapable in semantic ones. Some options may use other system-specified constraints not included in our work. Since our purpose of constraints inference is to generate misconfigurations and evaluate the system reactions instead of precise constraints analysis. We consider this limitation acceptable.

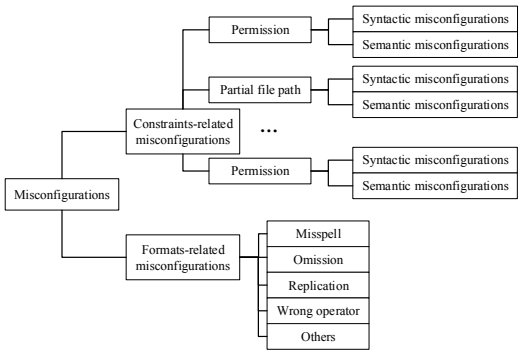


Figure 3: Misconfiguration generation rules

3 MISCONFIGURATION INJECTION

This section we answer two questions: How we use these constraints to generate misconfigurations and how we test system reactions with misconfigurations. To address these problems, we propose a tool, called ConfTest, to conduct misconfiguration injection and make analyses on system reactions to a variety of misconfigurations.

3.1 Misconfigurations Generation

To generate misconfigurations for injection, we defined some rules in ConfTest based on the constraints. Firstly, ConfTest parses configuration files into structured data for easily manipulating the configuration with predefined rules, then modifies original data to generate misconfigurations. Finally, these modified data are assembled to new configuration files with misconfigurations. As illustrated in Figure 3, the rules we used can be classified into two main categories: constraints-related and formats-related.

In constraints-related rules, ConfTest violates the constraints to generate misconfiguration. The constraints are defined as the description of what should be right for configuration option values both syntactically and semantically. We convert syntactic constraints into several conditions the value must satisfy, therefore, syntactic misconfigurations can be generated by violation of these conditions. For instance, as illustrated in Figure 4(a), 'Listen' in

Table 3: Syntactic constraints. Due to space limit, we show here simplified regular expressions and descriptions of syntactic constraints. Program analysis is used for inferring certain system specified constraints (e.g., Range of Count, the value set of Mode)

Type	Syntactic Constraints		
	Parttern	Element	Format of Element
File (Dierctory) Path	(/%S)+/?	S	[\w.-]+
Partial File (Directory) Path	%S(/%S)*	S	[\w.-]+
Domain Name	[%S_1]?://%S_2	S_1	(telnet|https|http|ftp)
		S_2	[a-zA-Z0-9.]+
IP Address	%D_1.%D_2.%D_3.%D_4	D_{1-4}	[0-255]
Email	%S_1@%S_2	S_1	(\w)+(\.\w+)*
		S_2	(\w)+((\.\w+)+)
Mode	%S	S	(value1|value2|value3)
Boolean	%S	S	(on|off|yes|no|true|false)
Language	%S	S	[a-zA-Z]{2}
MIME-types	%S	S	[\w/-.]+
Memory	%D %S	S	(K|M|G|T|KB|GB|TB|B)
		D	[min-max]
Time	%D %S	S	(s|min|h|d|ms)
		D	[min-max]
Speed Rate	%D %S	S	(bps|Mbps|Kbps)
		D	[min-max]
Count	%D	D	[min-max]
Fraction	%F	F	[min-max]
Port	%D	D	[0-65535]
Permission	%O	O	[0-777]
Username	%S	S	[a-zA-Z][a-zA-Z0-9]*
Password	%S	S	N/A
Filename	%S	S	[\w -]+.[\w -]+

Table 4: Semantic constraints

Option Type	Semantic Constraints	Resources
Path	The path should be existent	File System
URL	The URL should be reachable	Network
IP address	The IP should be accessible	Network
Email	The email should not be existent	Network
Domain name	The domain name should not be existent	Network
Port	The port should not be occupied	Services
Language	The language should be existent	Services
MIME type	The MIME type should be existent	Services
Memory	The memory should be less than available memory	Hardware
Speed rate	The speed rate should be less than available bandwidth	Hardware
Username	The username should be from root group	Security

Httpd is identified as PORT with syntactic constraints: value should be the integer between 0 and 65535. ConfTest can derive three conditions as shown in Figure 4(a). Therefore, by violating these conditions, syntactic misconfigurations are generated to test the upper and lower bound and the situation of the float type. Semantic constraints of type PORT is that option 'Listen' must not use the occupied port number. To violate this constraints, ConfTest acquires the occupied port from relevant execution information, such as '/etc/services' in system, and generates the semantic misconfigurations.

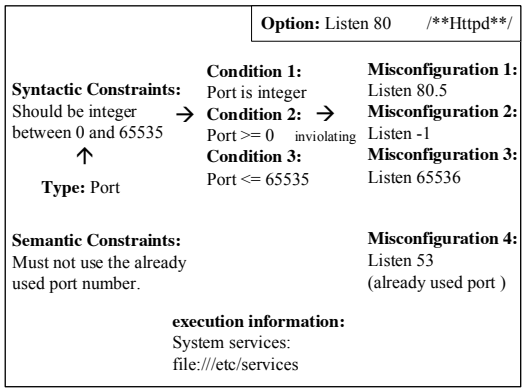
In formats-related rules, we consider the fact that configuration files often satisfy some specific format, such as some predefined module structure. Thus, as the example shown in Figure 4(b), ConfTest generates misconfigurations by simulating users' common mistakes, such as omission, misspelling, etc., while editing those complex configuration files.

3.2 Testing

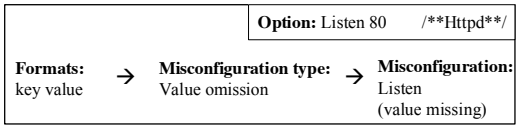
In order to evaluate system reactions better without the defection caused by the fact that even the same misconfiguration may cause different system reactions due to the different states of programs, ConfTest tests our misconfigurations in real world software environment by using software's own test infrastructure [2, 11, 14, 26] , including test cases and test oracles. ConfTest uses a sequence of test scripts (e.g. launching system, functional tests etc.) to simulate the administrators' behavior when faced with misconfigurations. Before the tests, ConfTest needs to check whether system can pass all these test cases, to make sure system is in the correct state.

Table 5: Environment related attributes for option type

Option Type	Environment-related Attribute	Type	Description
Path	Owner	String	Owner of Path
	Group	String	Group name of Path
	Permission	Octal	Permission of Path
	Contents	String	Contents of path
	HasSymLink	Bool	If has a symbol link
	Type	Enum	What's type of Path
URL	Forbidden	Bool	Is URL forbidden by firewall
	Type	Enum	What's type of URL
IP address	Forbidden	Bool	Is IP forbidden
	Type	Enum	What's type of IP
Port	User	String	What's the user of port
	Type	Enum	What's type of port
Language	IsSupported	Bool	Is Language supported by system
MIME type	IsSupported	Bool	Is MIME type supported by system
Memory	Allocation	Enum	Strategy of allocation
Username	GroupName	String	Group name of Username



(a) Constraints-related misconfigurations generation



(b) Formats-related misconfigurations generation

Figure 4: Challenges in diagnosing misconfiguration

Table 6: The proportion of options consistent with syntactic constraints

Software	Options	Satisfied
Httpd	564	466(82.6%)
MySQL	671	641(95.5%)
PostgreSQL	273	265(97.0%)
Yum	74	68(91.9%)
Total	1582	1440(91.0%)

Table 7: Systems in evaluation

Software	LoC	Options	Misconfigurations
Httpd	148K	30	236
MySQL	1.2M	26	255
PostgreSQL	757K	33	334
Yum	38K	26	244

Table 8: System reactions classification

Whether passes all tests	Whether has misconfiguration related exception information	Whether locate the miscon-figurations	Abbr.
T	T	T	Type 1
T	T	F	Type 2
T	F	-	Type 3
F	T	T	Type 4
F	T	F	Type 5
F	F	-	Type 6

When a misconfiguration is tested, ConfTest first uses the faulty configuration file(s) to replace the old one(s), and then launches the target system. If system succeeds in startup, ConfTest would run test cases persistently until system failure happens or all test cases executed, ConfTest monitors all the system and console logs to record the system state. At last, ConfTest analyses these logs to evaluate the system reaction towards misconfigurations.

4 ANALYSIS OF RESULTS

In this section, we evaluate the reaction ability based on the result from misconfiguration injection of ConfTest. Table 7 illustrates the studied software and their options. Generating and testing misconfigurations for all these options will lead to explosive exponential growth of computational complexity. To avoid this problem, we apply stratified random sampling method to options according to their types. Specifically, all the options are divided into subgroups according to the classification in Figure 2. For each option type, we randomly sampled options with fraction proportional to that of the total population. Options of different types varies greatly in system configuration, our sampling will ensure that estimates can be made with equal accuracy in different types of the configuration, and that comparisons of samples with different types can be made with equal statistical power. As shown in Table 7, we finally sampled 115 options and generated 1069 misconfigurations.

Table 9: Overall results of system reactions

Abbr.	Yum	Httpd	PostgreSQL	MySQL	Sum	Ratio
Type 1	0	0	0	11	11	1.03%
Type 2	4	0	0	0	4	0.37%
Type 3	127	65	155	105	452	42.28%
Type 4	82	113	176	70	441	41.25%
Type 5	3	8	3	57	71	6.64%
Type 6	28	50	0	12	90	8.42%
Sum	244	236	334	255	1069	100%

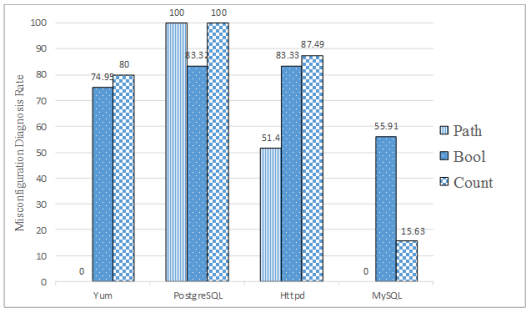


Figure 5: Misconfiguration diagnosis rate of different systems with 3 kinds of misconfiguration

4.1 System Reaction Ability Analysis

To evaluate the system reactions of misconfigurations, we classify the reactions into 6 types listed in Table 8. Given a misconfiguration under tested, we first check whether the system can pass the test suites we mentioned in Section 3.2. Secondly, we manually check all the log messages recorded under the test suites to find whether there is exception information related to misconfigurations. If so, lastly, we check whether this information could locate the misconfiguration. For example, Type1 reaction means that the system with the misconfiguration passed all the test cases without any failure, but we found exception information related to misconfigurations among the logs, furthermore, the exception information we found could explicitly locate the misconfigurations we injected. In this section, we use the abbreviations in Table 8 to represent the results of tests.

The overall results can be found in Table 9. To better understand the root causes of different system reactions, we further analyze each type of reactions. Type 1 (1.03%) and Type 2 (0.37%) reactions rarely happened, and only occurred in MySQL and Yum in our test. Type 1 reactions are resulted from overruling to invalid values during the assignments. For example, in MySQL, if option "table_open_cache" was misconfigured with an invalid number, MySQL would reassign this option to a normal one. Type 2 reactions have exception feedbacks, but still pass the functional tests (e.g., Yum prints logs like could not open particular file or directory during running but did not fail in the tests). Type 3 reactions are relatively common (42.28%), i.e., passing all the tests without throwing any exception. There are two occasions, one is the robust design that systems can legalize these misconfigurations. For instance, PostgreSQL allows both "key value" and "key = value" formats, which would avoid formats-related misconfigurations. In another case, there may be latent configuration errors (LC errors) [22]. Such options are not checked during initialization, hence we used various test cases to expose as many errors as we could. Type 4 reactions (41.25%) explicitly located the misconfigurations with line number or name of the option in configuration file. Most of these reactions happened during configurations checking, primarily in system startup. Type 5 reactions (6.64%) triggered the exceptions but failed to locate misconfigurations, mainly because options were not checked or checking condition failed in capturing the error. Even though being detected on exception, these reactions may be obscure or mislead users in diagnosis. Give an example, When Httpd was misconfigured incidentally by adding option "Listen 80" twice in configuration file, logs after failure printed that "Address already in use" and "Could not bind to address" which may mislead users. Type 6 reactions (8.42%) terribly obstructed the diagnosing. These reactions (e.g. crashes, hangs, silent failures) were caused by improper exception handling or lacking configuration checking.

4.2 Misconfiguration Diagnosis Analysis

In this section, we evaluate the reactions to different type of misconfigurations. As illustrated in Figure 5, we analyze three widely used option types, i.e., Path, Boolean and Count. We use misconfiguration diagnosis rate to evaluate the system reactions, which is the percentage of reactions that can locate the misconfigurations (Type 4) after system failures. We can see that MySQL and Yum failed to locate Path related misconfigurations. Compared with Path, Boolean misconfiguration is easier to localize. The diagnosis rate of Httpd, Yum, and PostgreSQL reaches 70%, while only 55.91% for MySQL. This is mainly because MySQL only reported the exception captured in systems without any location information about misconfiguration. Diagnosis rate of Count are even higher than those of Boolean in Httpd (87.49%), Yum (80%) and PostgreSQL (100%). Similar to the reason in Boolean misconfiguration, MySQL still has a low proportion (15.63%) in this respect.

Among these three types, Path misconfigurations are the hardest to be diagnosed by systems, even the experienced developers may fail to gracefully handle these misconfigurations. However, it is not impossible but requiring much more efforts to help diagnose these misconfigurations (e.g., PostgreSQL checks each configuration's syntax after startup). Misconfigurations of types with simple constraints (e.g., Boolean, Count, Mode) have a high diagnosis rate, mainly for the reason that the verifying these types' constraints can be easily implemented. Thus, using more simple constraints options in configuration is a highly recommended way for developers to reduce potential misconfigurations. What's more, users require more reasons than symptoms of failures (e.g., exceptions) in the systems. We recommend developers to point out the root causes instead of only recording what happened in the system and console log messages.

4.3 Capability Analysis

In this section, we evaluate the ConfTest's capability of finding bad system reactions (Type 5 and Type 6) by comparing ConfErr [9]. We choose ConfErr in our comparison for the reason that ConfErr

Table 10: Capability of ConfTest and ConfErr in finding bad system reactions

	# of undiagnosed misconfigurations / # of injected misconfigurations				
	Httpd	MySQL	PostgreSQL	Yum	Total
ConfTest	38/106 (35.85%)	60/169 (35.5%)	3/119 (2.52%)	23/101 (22.78%)	124/495 (25.05%)
ConfErr	30/183 (16.39%)	7/170 (4.12%)	2/105 (1.90%)	18/194 (9.28%)	57/652 (8.74%)

is not guided by constraints, it only generates misconfigurations by making generic alternations to valid configuration options (e.g., omissions, substitutions, and case alternations of characters).

In Table 10, we calculate the proportion of the undiagnosed misconfigurations (i.e., misconfigurations of which systems fail to find the root causes after failures, resulting in Type 5 and Type 6 reactions) generated by two tools. These undiagnosed misconfigurations could have been potentially avoided if ConfTest or ConfErr had been used to evaluate the system reactions and harden the system against misconfigurations. Our results show that, ConfTest found more undiagnosed misconfigurations in all 4 systems than ConfErr, and undiagnosed ones account for 25.05% of all while only 8.74% for ConfErr. The inefficiency in ConfErr is mainly because state-of-art software can easily detect the violation of configuration formats by checking configurations files. However, diagnosing constraints-related misconfigurations require not only domain knowledge but also environment information, which enable ConfTest to find the potential bad reactions of systems.

4.4 Threats to Validity

There are several major threats to the validity of our evaluation. (1) Although the software systems we studied in Section 2 are mature and large, our classification based on these systems may not be representative. In our work, to verify the generalization of our classification and avoid the overfitting, the configuration options we used in classification are from 8 open source software, while the invalidation in Section 2.1 are from another 4 open source software. (2) We only considered a subset of system options in our evaluation. There might be other configuration options, some hidden ones even not listed in the documentation, which may alter our results. However, all the options we chose are the frequently used ones and from existing configuration files, thus we believe that these configuration options are representative for the options in most cases. (3) All the results of our evaluation are based on system and console logs we recorded. Manually checking these information may introduce errors, so we have double checked all the results to ensure the correctness. (4) There may be some other misconfigurations not included in our works, in the future, we plan to reduce this threat further by analyzing more bug reports from open source and commercial software projects.

5 EXPERIENCE AND PRACTICE

In this section, we mainly talk about some advices or practices based on what we have observed from evaluated systems in misconfiguration issues.

Avoiding Inconsistency: We observed some good habits in handling unit inconsistency. For example, in PostgreSQL, numbers are assigned to options with units (e.g. max_stack_depth = 100kB) which clarifies users' confusion. To the contrary, lack in such suffixes or necessary explanations in configuration may trap administrator into making misconfigurations.

Making Configuration Simple and Easy: User-friendly design in configurations is also important for reduce the incidence of misconfigurations. In our research, we found many obscure options in these software documents. A good practice is that hiding these options from users, and only providing basic options instead. Like these practices, software system configuration should have different options and is targeted to different customers.

Early Checking: After checking the source code of these evaluated systems, we found that PostgreSQL processes the configuration file through consistent interface. During this process, values of each option are enforced to go through the checking, and diagnostic messages are printed in the situation on exception. Early checking helps users effectively detect the misconfigurations in the startup.

Friendly Comment: In configuration, PostgreSQL explicitly informs users of usage of specified options by comments (e.g., "#1s-600s", "#defaults to 'localhost' "; "use '*' for all") as guidance or handbooks. Such comments also can be found in many other systems. Adequate comments could guide users to have options configured correctly.

6 RELATED WORK

To address misconfiguration problems, many research efforts have been made by focusing on detecting [17, 21, 27] and troubleshooting [1, 4, 5, 10, 15, 20, 28] misconfigurations. Although it's helpful for people to understand and address misconfigurations, only few efforts [9, 23] focus on evaluating system reaction ability of misconfigurations, also known as configuration testing [24].

Configuration Testing: To improve systems reliability, researchers evaluate software systems by testing system reactions to configuration errors. We refer to such testing efforts as configuration testing. ConfErr [9] plays the role of a pioneer in configuration testing, it uses human errors model to simulate human mistakes (e.g. typo, copy-paste mistake, and other generic alternations.) in configuration. However, ConfErr is not guided by the configuration constraints, and it can only generate deficient misconfigurations, which impedes the analyses on system reactions. SPEX [23] goes a step further, it infers constraints through program analysis to find vulnerabilities and error-prone design by injecting constraints-violated misconfigurations. However, its coarse-grained constraints resulted in the poor diversity of generated misconfigurations. Based on fine-grained constraints of option types, ConfTest could generate comprehensive misconfigurations, which means the system reaction ability of misconfigurations could be effectively evaluated.

Misconfiguration Detecting and Troubleshooting: Misconfiguration detection [17, 21, 27] refers to checking the configuration

options before the misconfigurations manifest, while the misconfiguration troubleshooting [1, 4, 5, 10, 15, 20, 28] is carried out afterwards. EnCore [27] uses machine-learning method to infer the configuration rules between applications and executing environment, which can be used for detecting misconfigurations. X-ray [4] proposes a technique for automatically diagnosing the root causes of performance problems. Our work is complementary to these works. The major part of our works is generating comprehensive misconfigurations through constraints, these constraints can be used as references for developers and researchers to detect the misconfigurations. Meanwhile, misconfiguration troubleshooting can also benefit from our evaluation on system reactions by analyzing the characteristics of them.

Others: There are also some other researches on misconfiguration issues. Rabkin and Katz [16] proposed a taxonomy of configuration options after study the options in several Java applications. Our classification of configuration option is mainly different from theirs in objectives, they are aimed at automatically extracting options from the source code, while we intend to infer constraints for each type. Therefore, we need a more fine-grained classification to infer various constraints as much as possible. ConfDiagDetector [29] focuses on detecting inadequate diagnostic messages of misconfigurations. Yin et al. [25] conducted a real-world misconfiguration characteristic study on 546 real world misconfigurations. Both of them are based on the system reactions and would benefit from our work.

7 CONCLUSION

Misconfigurations have become the major cause of software failures. In this paper, we proposed a misconfiguration-injection method to evaluate the system reactions ability. To help infer constraints, we studied 8 mature open-source systems to summarize a fine-grained classification of option types. Based on this classification, we are able to summarize and extract fine-grained options constraints to generate misconfigurations. We implemented a tool named ConfTest to conduct misconfiguration injection and make analyses on system reaction abilities to variety of misconfigurations. Our evaluation results show that our option classification covers over 96% of 1582 options from Httpd, Yum, PostgreSQL and MySQL. Our constraint is more fined-grained and the accuracy is more than 90% through manual verification. We compared the capability in finding bad system reactions between ConfTest and ConfErr, the results show that the ConfTest can find nearly 3 times the bad reactions found by ConfErr.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation (61690203, 61532007) and National 973 Program (2014CB340703) of China.

REFERENCES

[1] Bhavish Aggarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N Padmanabhan, and Geoffrey M Voelker. 2009. NetPrints: diagnosing home network misconfigurations using shared knowledge. In *Usenix Symposium on Networked Systems Design and Implementation*. 349–364.

[2] Apache. 2017. Apache HTTP Test Project. Retrieved January 29 from http://httpd.apache.org/test/. (2017).

[3] Computing Research Association. 2003. Grand research challenges in information systems. *A Conference Series on Grand Research Challenges in Computer Science and Engineering*. (2003).

[4] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: automating root-cause diagnosis of performance anomalies in production software. In *Usenix Conference on Operating Systems Design and Implementation*. 307–320.

[5] Mona Attariyan and Jason Flinn. 2008. Using Causality to Diagnose Configuration Bugs.. In *Usenix Technical Conference, Boston, Ma, Usa, June 22-27, 2008. Proceedings*. 281–286.

[6] Mona Attariyan and Jason Flinn. 2010. Automating configuration troubleshooting with dynamic information flow analysis. In *Usenix Conference on Operating Systems Design and Implementation*. 1–11.

[7] L Barroso, J Clidaras, and U Hoelzle. 2009. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. 8, 3 (2009), 154.

[8] Jim Gray. 1985. Why do computers stop and what can be done about them. *Technical Report TR-85.7* 30, 4 (1985), 88–94.

[9] L Keller, P Upadhyaya, and G Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *IEEE International Conference on Dependable Systems and Networks with Ftcs and DCC*. 157–166.

[10] James Mickens, Martin Szummer, and Dushyanth Narayanan. 2007. Snitch: interactive decision trees for troubleshooting misconfigurations. In *Usenix Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*. 8.

[11] MySQL. 2017. The MySQL Test Framework. Retrieved January 29 from https://dev.mysql.com/doc/mysqltest/2.0/en/. (2017).

[12] Kiran Nagaraja, F Oliveira, Ricardo Bianchini, Richard P Martin, and Thu D Nguyen. 2004. Understanding and dealing with operator mistakes in internet services. In *Conference on Symposium on Opearting Systems Design & Implementation*. 61–76.

[13] David Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. Why Do Internet Services Fail, and What Can Be Done About It?. In *Conference on Usenix Symposium on Internet Technologies and Systems*. 1–1.

[14] PostgreSQL. 2017. PostgreSQL 9.6.1 Documentation. Retrieved January 29 from https://www.postgresql.org/docs/9.6/static/pgbench.html. (2017).

[15] Ariel Rabkin and Randy Katz. 2011. Precomputing possible configuration error diagnoses. In *Ieee/acm International Conference on Automated Software Engineering*. 193–202.

[16] Ariel Rabkin and Randy Katz. 2011. Static extraction of program configuration options. In *International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , Hi, Usa, May*. 131–140.

[17] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury. 2009. Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *International Conference on Autonomic Computing, Icac 2009, June 15-19, 2009, Barcelona, Spain*. 169–178.

[18] Y Sverdlik. 2012. Microsoft: misconfigured network device led to azure outage. Retrieved January 29 from http://www.datacenterdynamics.com/focus/archive/2012/07/microsoft-misconfigured-network-device-led-azure-outage. (2012).

[19] A Team. 2011. Summary of the amazon ec2 and amazon rds service disruption in the us east region. Retrieved January 29 from http://aws. amazon. com/message/65648. (2011).

[20] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi Min Wang. 2004. Automatic misconfiguration troubleshooting with peerpressure. In *Conference on Symposium on Opearting Systems Design & Implementation*. 17–17.

[21] Yi Min Wang, Chad Verbowskia, John Dunagana, Chenb Yu, Helen J. Wanga, Chun Yuanb, and Zhangb Zheng. 2003. STRIDER: A Blackbox, State-based Approach to Change and Configuration Management and Support. In *Usenix Conference on System Administration*. 159–172.

[22] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. (2016).

[23] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Twenty-Fourth ACM Symposium on Operating Systems Principles*. 244–259.

[24] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *Acm Computing Surveys* 47, 4 (2015), 1–41.

[25] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October*. 159–172.

[26] Yum. 2017. QA:Testcase Yum basics. Retrieved January 29 from http://fedoraproject.org/wiki/QA:Testcase_Yum_basics. (2017).

[27] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: exploiting system environment and correlation information for misconfiguration detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–700.

[28] Sai Zhang and Michael D. Ernst. 2013. Automated diagnosis of software configuration errors. In *International Conference on Software Engineering*. 312–321.
[29] Sai Zhang and Michael D Ernst. 2015. Proactive detection of inadequate diagnostic messages for software configuration errors. (2015).