2018 IEEE International Conference on Software Quality, Reliability and Security

MisconfDoctor: Diagnosing Misconfiguration via Log-based Configuration Testing

Teng Wang*, Xiaodong Liu*, Shanshan Li*, Xiangke Liao*, Wang Li*, Qing Liao†
*College of Computer Science, National University of Defense Technology, Changsha, China
†Department of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China
Email: {wangteng13, liuxiaodong, shanshanli, xkliao, liwang2015} @nudt.edu.cn, liaoqing@hit.edu.cn

***Abstract*—As software configurations continue to grow in complexity, misconfiguration has become one of major causes of software failure. Software configuration errors can have catastrophic consequences, seriously affecting the normal use of software and quality of service. And misconfiguration diagnosis faces many challenges, such as path-explosion problems and incomplete statistical data. Our study of the log that is generated in response to misconfigurations by six widely used pieces of software highlights some interesting characteristics. These observations have influenced the design of MisconfDoctor, a misconfiguration diagnosis tool via log-based configuration testing. Through comprehensive misconfiguration testing, MisconfDoctor first extracts log features for every misconfiguration and builds a feature database. When a system misconfiguration occurs, MisconfDoctor suggests potential misconfigurations by calculating the similarity of the new exception log to the feature database. We use manual and real-world error cases from Httpd, MySQL and PostgreSQL in order to evaluate the effectiveness of the tool. Experimental results demonstrate that the tool's accuracy reaches 85% when applied to manual-error cases, and 78% for real-world cases.**

***Keywords*-Misconfguration; Exception log; Configuration testing**

I. Introduction

With the widespread use of the internet and computer technology, large-scale software systems have become an indispensable part of modern society. However, due to the expanding scale of the software and the increasing demands for customizability, software developers need to provide a large number of configuration options. While more configuration options allow users to configure and manage their software more conveniently and flexibly, they are more likely to cause configuration errors. As software configurations continue to grow in complexity, misconfiguration has become one of the major causes of software failure [1-3]. Yin et al. [4] report that 27% of customer support cases at a major storage company are related to configuration issues. Oppenheimer et al. [5] find that more than 50% of system operation and maintenance failures are due to misconfiguration and can result in the inability to use system services.

Software configuration errors can have catastrophic consequences, seriously affecting the normal use of the software and the quality of service. For example, Facebook went down for 2.5 hours in 2010 and was inaccessible for up to five million people because of configuration errors [6]. The serious consequences of configuration errors make it imperative and vital that research into the diagnosis and repair of misconfigurations be undertaken.

Recent research into software configuration diagnosis consists mainly of program analysis and statistics-based approach. The program analysis methods [7-10] analyze the corresponding statements given by the configuration variables and the execution path and thereby infer which configuration option may have caused the software system error. However, such methods are deficient in terms of their efficiency and scalability, due to the path explosion problem. The statistics-based approach can be divided into the rule-based approach [11-14] and the signature-based approach [29-32]. The rule-based approach learns rules from the correctly functioning system; if a configuration setting does not satisfy these learned rules, it will be labeled as a misconfiguration. This approach has a higher false-positive rate because it is difficult to guarantee the completeness of the rules and avoid learning inaccurate rules. Signature-based approaches compare the signature of the failure execution with normal signatures in order to reason out configuration errors [33]. When some emerging features replicate or largely resemble the signature of a known failure, we can probably assume that the current software contains the failure. These methods are limited by the size and comprehensiveness of the reference signatures, which can greatly affect their accuracy.

The challenges of signature-based methods are mainly twofold. Firstly, how best to choose the appropriate information as a signature is a non-trivial issue, as it needs to accurately describe the status of the software runtime and capture the behavior of the misconfiguration. Wang et al. [32] capture the Windows Registry entries as the signature of the failure execution, which limits their application to software running on operating systems like Linux and UNIX. Yuan et al. [29] collects system call traces and uses the system behavior information as the signature. However, the argument against using system behavior information to characterize problems is that such a signature is usually less accurate and less direct compared to a program-generated signature [29].

Secondly, the scale and complexity of a software system significantly enlarges the size of a signature database, greatly increasing the difficulties experienced when building a comprehensive signature database. Jiang's study [15][16]

978-1-5386-7757-5/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS.2018.00014

manually collected actual customer cases in two-years' worth of commercial storage system reports. Even though this resulted in extensive signature data collection through a very high workload, it still cannot possibly reflect all the potential misconfiguration cases that could occur within the target software.

To address the above challenges, we present MisconfDoctor, a misconfiguration diagnosis tool that uses log-based configuration testing. We chose the software log to describe the system's signatures, since a log can be a good representation of the software's behavior and status, and can thereby greatly aid the failure diagnosis process [36][37]. We carry out in-depth research on the log that is generated in response to misconfigurations by six open-source pieces of software, and find that the exception log sequence, which appears when the software fails, usually contains some special features. For example, in comparison to the normal log sequence, the exception log sequence may increase one or more logs in certain formats by describing potential failure information; additionally, it may miss part of the log in the normal log sequence. We collect such exception log features as signatures to help the misconfiguration diagnosis. MisconfDoctor explores our previous work ConfTest [21] in order to generate comprehensive misconfigurations and utilize a method of configuration testing in order to obtain the log sequence feature and build the signature database.

The contributions of this paper are as follows:

- We collect software logs and extract the exception log features in order to build a signature database for misconfiguration troubleshooting. A configuration testing method is used to guarantee the comprehensiveness of our proposal.
- We design and implement MisconfDoctor, a tool for diagnosing misconfigurations through log-based configuration testing. We calculate the similarity between the target log sequence and the exception log feature and suggest the misconfiguration culprit.
- We use manual and real-world error cases from Httpd, MySQL and PostgreSQL in order to evaluate the tool's effectiveness. Experimental results demonstrate that the accuracy reaches 85% for the manual error cases and 78% for real-world cases.

The remainder of this paper is organized as follows. We present our study of exception logs in Section 2, before introducing the architecture of MisconfDoctor in Section 3 and its design in Section 4. The evaluation is found in Section 5, we present related work in Section 6 and we conclude our work in Section 7.

II. Study of Features in Exception Logs

In this section, we will first introduce observations concerning the characteristics of software logs, before discussing the features of exception logs.

A. *Observations on the Characteristics of Logs*

A software log is an important guide for failure diagnosis. It can record dynamic information when the program is running and helps the developer to analyze and reproduce the errors. The standard, sufficient and useful logs are very significant in software failure diagnosis, as they can effectively improve the efficiency of both troubleshooting and recovery. However, the logs generated by many examples of popular software (including both open-source software and commercial software) partly come with insufficient guidance, which makes it difficult for users to directly use the log statement to diagnose the cause of the failure.

We investigate the source code and logs of six pieces of software (Httpd, MySQL, PostgreSQL, Redis, Nginx and OpenSSH) in order to find the reason why the current trend in software is to make it difficult to directly help users diagnose failure. This trend manifests as follows:

- The log description is too professional or vague, or
- The log contains misleading information.

The log messages provided represent the execution process and system status of the software in a serialized form. When software fails due to different misconfigurations, the log sequence presents specific pattern characteristics, which can effectively aid in diagnosis. We find that, under certain inputs, the software execution path and log output can be fixed.

1) Vague log description: Due to the lack of unified log standards and the fact that log statements can be influenced by the habits of software developers, the software log contains many vague descriptions or specialized information (such as program exception codes, which only software developers can understand). Although the contents of these logs cannot help us diagnose errors directly, the pattern of the log is helpful.

Fig. 1(a) provides an example of PostgreSQL's log. We change the configuration option hba_file to "/var" and the PostgreSQL server fails to start. The log gives detailed information, such as "could not open configuration file '/var': Permission denied" and "could not load pg_hba.conf". However, it does not indicate the option hba_file, meaning that the average user would not be able to find the cause of the error. That being said, this log sequence appears in a pattern that reflects the execution path and status of the software.

We analyze the source code of PostgreSQL, as shown in Fig. 2. The Postmaster process is the first process in PostgreSQL, and it performs the initialization work of loading the configuration file for client authentication via load_hba(). The function load_hba() uses the method AllocateFile() to open HBAFileName, which is the value of option hba_file. When the file fails to open, load_hba() will output the given log and then PostmasterMain() will also output the given log. Such a log sequence implicitly reveals that when there is an error in the option hba_file the software fails to start

```
systemd[1]: Starting PostgreSQL database server...
pg_ctl[593]: LOG:  could not open configuration file "/var": Permission
denied
pg_ctl[593]: FATAL:  could not load pg_hba.conf
pg_ctl[593]: pg_ctl: could not start server
systemd[1]: Failed to start PostgreSQL database server.
systemd[1]: Unit postgresql.service entered failed state.
systemd[1]: postgresql.service failed.
```

(a) Log of PostgreSQL

```
Starting The Apache HTTP Server...
httpd[129549]: Syntax error on line 7 of /etc/httpd/conf/httpd.conf: Could
not open config directory /abc/conf.modules.d: No such file or directory
systemd[1]: httpd.service: main process exited, code=exited,
status=1/FAILURE
kill[129551]: kill: cannot find process ""
systemd[1]: httpd.service: control process exited, code=exited status=1
systemd[1]: Failed to start The Apache HTTP Server.
systemd[1]: Unit httpd.service entered failed state.
```

(b) Log of Httpd

Figure 1. Examples of exception logs

```
/* postgresql-9.3.1/backend/postmaster/postmaster.c */
void PostmasterMain(int argc, char *argv[])
…
        conffile = ap_server_root_relative(cmd->pool, name);
        if (!load_hba()){
             ereport(FATAL,
                           (errmsg("could not load pg_hba.conf")));}
}

/* postgresql-9.3.1/backend/libpq/hba.c */
load_hba(void){
…
        file = AllocateFile(HbaFileName, "r");
        if (file == NULL){
             ereport(LOG, (errcode_for_file_access(),
                           errmsg("could not open configuration file
                                         \"%s\": %m", HbaFileName)));}
}
```

Figure 2. Source code snippet of PostgreSQL

```
/* httpd-2.4.24/server/core.c */
include_config (cmd_parms *cmd, void *dummy, const char* name){
…
        conffile = ap_server_root_relative(cmd->pool, name);
…
        ap_process_fnmatch_configs(cmd->server, conffile, &conftree,
                    cmd->pool, cmd->temp_pool, optional);
…
}

/* httpd-2.4.24/server/config.c */
ap_process_resource_config_nofnmatch(…){
…
        rv = apr_dir_open(&dirp, path, ptemp);
        if (rv != APR_SUCCESS) {
        return apr_psprintf(p, "Could not open config
                directory %s: %pm", path, &rv); }
}
```

Figure 3. Source code snippet of Httpd

and outputs the log sequence as shown above.

2) Log with misleading information: We find that the error message described by the software log is not always where the software really goes wrong. That is, the potential misconfiguration of the software is latent and bursts out when other options are used. Additionally, if the software cannot solve the misconfiguration effectively, then the log information can be misleading. In this case, while the software log cannot help us diagnose problems directly, the pattern is still helpful.

Fig. 1(b) provides an example of an Httpd log. We set the option ServerRoot to "/abc" and start the Httpd server. The log points to "Syntax error on line 7 of /etc/httpd/conf/httpd.conf: Could not open config directory /abc/conf.modules.d". After checking the file httpd.conf, we know that the log indicates that there is a problem with the option "Include". However, it is not the root cause.

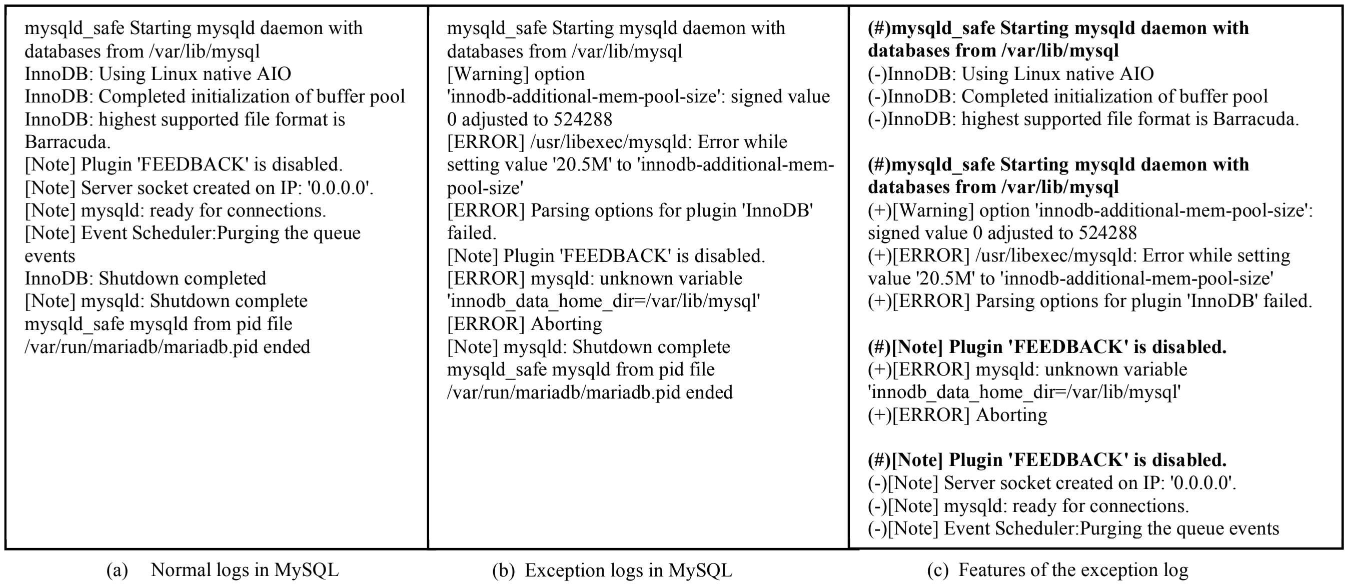
We analyze the source code of Httpd, as shown in Fig. 3. The software uses the function include_config() to load the option Include, which uses the value ServerRoot as part of the path. Thus, ServerRoot's error caused the failure when loading Include. The software cannot analyze the root cause of the error itself, so the given log cannot directly help the user to diagnose the problem. However, this log sequence does shows the system behavior when ServerRoot errors occur, which is helpful for diagnostics.

B. Observations on the Feature in Exception Logs

In the previous part, we learned that log sequences show a specific pattern in the case of software failure, and that the feature of the exception log can help us to diagnose errors. That is to say, when the software fails, if the new exception logs contain the known exception feature or are largely similar to it, we can assume that the software may contain the corresponding known failures.

We study the logs of six pieces of software with misconfigurations during testing and questions about configuration errors from StackOverflow [19] and ServerFault [20] (two online Q&A websites focusing on computer knowledge). Compared with the log sequence that results from the correct execution, we find that the exception logs will show the following features:

- The addition of one or more logs in certain formats
- Missing parts of logs in the normal log sequence



```
mysqld_safe Starting mysqld daemon with
databases from /var/lib/mysql
InnoDB: Using Linux native AIO
InnoDB: Completed initialization of buffer pool
InnoDB: highest supported file format is
Barracuda.
[Note] Plugin 'FEEDBACK' is disabled.
[Note] Server socket created on IP: '0.0.0.0'.
[Note] mysqld: ready for connections.
[Note] Event Scheduler:Purging the queue
events
InnoDB: Shutdown completed
[Note] mysqld: Shutdown complete
mysqld_safe mysqld from pid file
/var/run/mariadb/mariadb.pid ended
```

(a) Normal logs in MySQL

```
mysqld_safe Starting mysqld daemon with
databases from /var/lib/mysql
[Warning] option
'innodb-additional-mem-pool-size': signed value
0 adjusted to 524288
[ERROR] /usr/libexec/mysqld: Error while
setting value '20.5M' to 'innodb-additional-mem-
pool-size'
[ERROR] Parsing options for plugin 'InnoDB'
failed.
[Note] Plugin 'FEEDBACK' is disabled.
[ERROR] mysqld: unknown variable
'innodb_data_home_dir=/var/lib/mysql'
[ERROR] Aborting
[Note] mysqld: Shutdown complete
mysqld_safe mysqld from pid file
/var/run/mariadb/mariadb.pid ended
```

(b) Exception logs in MySQL

```
(#)mysqld_safe Starting mysqld daemon with
databases from /var/lib/mysql
(-)InnoDB: Using Linux native AIO
(-)InnoDB: Completed initialization of buffer pool
(-)InnoDB: highest supported file format is Barracuda.

(#)mysqld_safe Starting mysqld daemon with
databases from /var/lib/mysql
(+)[Warning] option 'innodb-additional-mem-pool-size':
signed value 0 adjusted to 524288
(+)[ERROR] /usr/libexec/mysqld: Error while setting
value '20.5M' to 'innodb-additional-mem-pool-size'
(+)[ERROR] Parsing options for plugin 'InnoDB' failed.

(#)[Note] Plugin 'FEEDBACK' is disabled.
(+)[ERROR] mysqld: unknown variable
'innodb_data_home_dir=/var/lib/mysql'
(+)[ERROR] Aborting

(#)[Note] Plugin 'FEEDBACK' is disabled.
(-)[Note] Server socket created on IP: '0.0.0.0'.
(-)[Note] mysqld: ready for connections.
(-)[Note] Event Scheduler:Purging the queue events
```

(c) Features of the exception log

Figure 4. Examples of log sequences and features of the exception logs

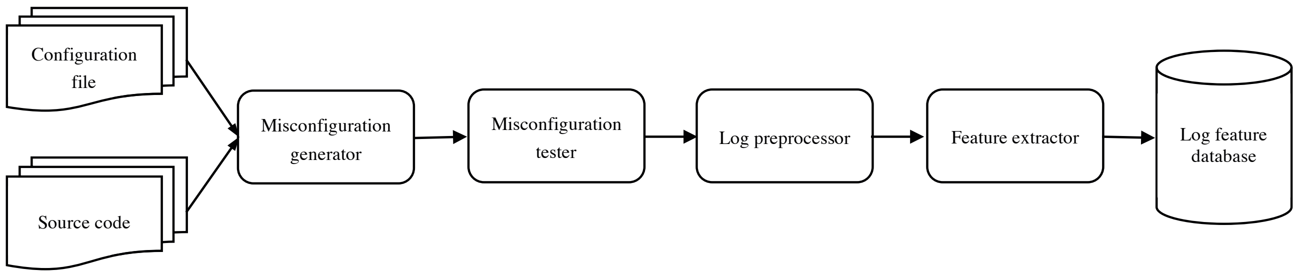


Figure 5. Architecture of the learning component of MisconfDoctor

1) The addition of one or more logs in certain formats: The log sequence indicates the execution path of the software. When the software handles a module abnormality, the log system will print an additional notice, warning, error or other information to indicate that these are not operating correctly. That is, it shows that the program has deviated from the normal execution path specified by the system and so it will add more logs in a specific format. The difference in these logs is obvious. Fig. 4(a) shows the log when MySQL starts normally, and Fig. 4(b) shows the log when the service fails due to a configuration error. The additional three logs (Warning, ERROR, ERROR) in the exception logs indicate the fault information provided by the software, which we call one feature of the known misconfiguration. Following the pattern of the code changes, we present the log difference as log changes, as shown in Fig. 4(c). The symbol "#" represents a common log and the symbol "+" represents the additional log when misconfiguration occurs.

2) Miss part of logs in the normal log sequence: When an exception occurs somewhere in the program execution, the execution path deviates from the normal path. As a result, multiple logs go missing from the normal log sequence. Missing logs are also an important feature of exceptions. Similarly, we can present log differences as log changes, obtaining four features as shown in Fig. 4(c). The symbol "-" represents the missing log when misconfiguration occurs. Each "+" feature and "-" feature are found under one "#" log.

III. The Architecture of MisconfDoctor

This section describes the architecture of MisconfDoctor, which diagnoses misconfigurations through the analysis of exception log features resulting from misconfiguration testing. MisconfDoctor contains two components: the learning component and the diagnosing component.

The goal of the learning component is to obtain the exception log features for the target software. These extracted

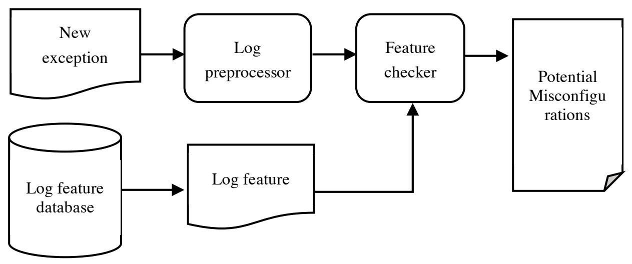


Figure 6. Architecture of the diagnosing component of MisconfDoctor

features are further applied by the diagnosing component in order to diagnose the configuration errors. The learning component first generates comprehensive misconfigurations for the target software and conducts tests in order to obtain the logs. Following this, it can extract log features for every misconfiguration. As illustrated in Fig. 5, the learning component has four main parts, as follows.

Misconfiguration generator: The diagnosis is based on our ability to get the log sequence information from the target software in response to various failures that are due to various configuration errors. In order to achieve this goal, we first need to generate a comprehensive set of misconfigurations, including both single-option and multi-option misconfigurations.

Misconfiguration tester: The tester injects every misconfiguration the generator creates into the original configuration file, carrying out the same test process. Following this, the tester records the misconfiguration (M) and its error logs (L), thus forming a tuple $< M, L >$. The aim is to get various tuples of $< M, L >$; as such, our tests need to have a high coverage of the target software.

Log preprocessor: The logs we obtain from the tests always contain a lot of redundant information, such as that regarding execution time and environmental variables. Before extracting the exception log features, we need to preprocess the logs into a standard format.

Feature extractor: Based on the observations discussed in Section 2, exception logs have some features compared with normal logs. The work of the extractor is to extract special features from every exception log sequence and save the tuples $< M, LF >$ of the misconfiguration (M) and its exception log feature (LF) into a log feature database.

The diagnosing component detects potential misconfigurations using the new exception logs and the log feature database. Exception logs will accompany any failure in the software. As depicted in Fig. 6, the component's input is the new exception logs and it outputs potential misconfigurations. When the software fails, then we can obtain the new exception logs. After log preprocessing takes place, the feature checker will check whether the log sequence contains any special features, which are saved in the log feature database. If the log sequence contains special features, the target software environment may contain relevant misconfigurations. The checker will then list the potential misconfigurations, following which the user can check for them conveniently.

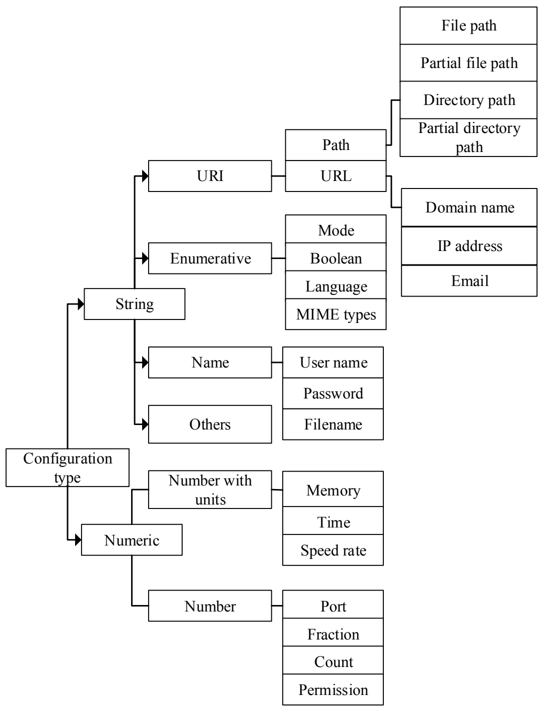


Figure 7. Type classification rule for configuration options [21]

IV. Design And Implementation

A. Misconfiguration Generator

In order to obtain comprehensive exception log sequences, we first need to generate comprehensive misconfigurations. We classify misconfigurations into two categories: single-option and multi-option misconfigurations.

Single-option misconfigurations just have mistakes in one option; these will be either format-related or constraint-related misconfigurations. Most software configurations usually come in key-value pairs. We simulate the mistakes people may make when using the configuration option, summarizing the format-related errors as Misspell, Omission, Replication, Case error and Wrong operator. Our previous work (ConfTest [21]) has completed the design of this method. In our previous study [21] we researched more than 2,000 configuration options in eight widely used pieces of software and classified these options into fine-grained types. For every option, we extracted the corresponding syntactic and semantic constraints before generating constraint-related errors. The fine-grained type classification is shown in Fig. 7. ConfTest is sufficiently comprehensive for single-option misconfigurations.

```
/* postgresql-9.3.1/backend/postmaster/postmaster.c */
PostmasterMain(int argc, char *argv[])        {
…
        if (max_wal_senders >= MaxConnections){
                write_stderr("%s: max_wal_senders must be less than
                        max_connections\n", progname);
                        ExitPostmaster(1);
                }
}
```

(a) Example of value constraint

```
/* postgresql-9.3.1/backend/access/transam/xlog.c */
RecordTransactionCommit() {
…
        if (…&& enableFsync)
                        pg_usleep(CommitDelay);
}
```

(b) Example of control constraint

Figure 8. Examples of Multi-option constraint

When analyzing multi-option misconfigurations, we generate misconfigurations using the constraints between two options. We study the source code of six pieces of software, as mentioned in Section 2, and find that multi-option constraints are shown to be value and control constraints that occur between two related options. “Value constraint” means that two options have a categorical value relationship (for example, “max_wal_senders” must be less than “max_connections” in PostgreSQL, as shown in Fig. 8(a).) “Control constraint” means that option B can work only when option A is in valid condition (for example, “pg_usleep(CommitDelay)” does not work if “enableFsync” is not turned on, as shown in Fig. 8(b).) Our previous work [22] can automatically extract the value constraints and control constraints in conditional branches of the source code. Our method is based on program analysis conducted on top of the LLVM compiler infrastructure [23]. On the basis of the constraints, we obey the value relationship and the control conditions. For example, we set “max_wal_senders” to be larger than “max_connections” in Fig. 8(a).

B. Misconfiguration Tester

The mission of a misconfiguration tester is to test system reactions under every misconfiguration the generator creates and record the system behavior and logs. In order to achieve high test coverage for the target software, we use the software’s own test infrastructure [24-26], including test cases and test oracles, in order to check whether the system fails under every misconfiguration condition. We consider the function tests and regression tests for the software to be comprehensive, with high coverage. We have therefore written a script tool that records both the additional content of the software log file after every test is executed and the corresponding configuration error.

Before the tests, we need to adjust all the configurations to the proper settings and check whether the system can pass all these test cases in order to make sure the software system is in the correct state. For every test case in the normal configuration, the tester also records the log sequence information, which is then referred to when extracting the features from the error log sequence.

We test only one misconfiguration each time. The tester injects the fault into the configuration file and then launches the target system. If the system fails to start up, that means the fault has influenced the initialization work. The tester will record the logs, following which there is no need to test this misconfiguration anymore.

We will then conduct every function test and check the system’s behavior according to the test oracles. When the software cannot pass a test, the tester will take the error log sequence (L) and misconfiguration (M) to form the tuple $< M, L >$ for the special test case. If the software passes the test, we note that the fault does not affect the special function or that it has been fixed by the software, and move on the next test case. After all the tests for every misconfiguration have been conducted, we have a collection of various tuples of $< M, L >$ for unpassed test cases.

C. Log Preprocessor

1) Log-text preprocessin: Normally, every sentence in the logs has redundant information; for example, the execution time, thread number and so on. This redundant information is mostly relevant to the execution environment and has a negative effect on feature extraction and diagnosis. As such, we need to perform standardized processing in order to obtain a uniform sentence.

In the first step, we use regular expressions to remove the execution time, such as “Jan 15 18:24:22”. Secondly, as the log sentence contains many program parameters that show the state of the system, it is necessary to turn the program parameters into a unified form. We use regular expressions to replace integer numbers with the word “integer”, float numbers with “float”, file paths with “file”, and directory paths with “directory”. Any email addresses are replaced by “email” and IP numbers with “ip-number”. We only convert the path that is contained in quotation marks because the path outside of these marks usually contains the program component path.

Thirdly, we remove the punctuation marks to make sure that every sentence only contains words. Fourthly, we change all the words into lowercase. Fifthly, we convert the forms of plurals and past tense into word prototypes using the Porter stemming algorithm [27]. After the canonicalization, the log sentence from MySQL, “170116 14:14:45 [ERROR] /usr/libexec/ mysqld: Error while setting value ‘20.5M’ to ‘innodb-additional-mem-pool-size’” is turned into “error usr ibexec mysqld error while setting value float to innodb-additional-mem-pool-size”.

Before Alignment	After Alignment
(1)“a b c d f h i”	(1)“a b c d _ f _ h i”
(2)“b c e f g i”	(2)“_ b c _ e f g _ i”

Figure 9. Example of sequence alignment

2) Noise filtering: After log-text preprocessing, we can remove the environment factor and special parameters in a single log sentence and thereby establish the standard log. However, the single log sentence may be environmentally sensitive noise. In order to determine whether this is the case, we ran the software twice more (on the same computer with the same test under the same configuration) and we found that we sometimes could not obtain two log sequences that were exactly the same. The differences in the log sequences may be due to the noise. Therefore, if we do not filter the noise, it will influence the features in the log sequence.

We do the same test 10 times, resulting in 10 log sequences for the same misconfiguration test. Most of the content is the same; only a small part is different. For every log sentence after canonicalization, we count how many times it occurs over the 10 sequences. If a sentence occurs in more than a threshold percentage of all the sequences, we regard it as a normal log sentence. Otherwise, it is considered to be noise. We set the threshold to 0.7 in our experiment. The method of selecting the threshold will be discussed in Section 5.3.

D. Feature Extractor

Based on the log-text preprocessing and noise filter, we are able to obtain standardized log sequences. We can then take both the log sequence produced in response to a correct execution and the exceptions in order to extract the features, as mentioned in Section 2. We take the log of the correct execution as the benchmark and compare the log sequence of the exceptions in serialization in order to ascertain the added and the missing logs.

We consider every sentence as a word, and the log sequence as a string. Two words are considered to be the same if their longest common substring accounts for 90% of the average length of the two words. The mission is to align the two sequences and collect the differences. We use a sequence alignment tool [28] to compare the differences. A sequence alignment algorithm tries to find the maximal similarity of two sequences and align the corresponding location for every word. Fig. 9 shows an example of the alignment of two strings; the underscores represent the missing word that is found when the string is compared to the other sequence.

Based on the sequence alignment algorithm, we can extract features for every exception log sequence. Algorithm 1 shows the process of extracting features. We take the last-matched sentence in both sequences and the missing subsequence from the standard sequence, using “(#)” and “(-)” as one feature and putting it into the feature set. Moreover, the additional subsequence from the exception sequence using “(+)” are also taken as one feature. We then establish several features of the exception log sequence. Finally, we save the tuples $< M, LF >$ of misconfiguration (M) and its exception log feature (LF) into the log feature database.

Table I
ALGORITHM 1: FEATURE EXTRACTION

```
Input:  the standardized StandardSequence(SSeq) and the standardized ExceptionSequence(ESeq)
Output: the feature set(FS)
1   sequence alignment for SSeq and ESeq
2   For sentence in ESeq:
3     If find a matched sentence in SSeq:
4       Add [(#) the previous matched sentence] to FS
5       Add [(-) missing subsequence] from SSeq to FS
6     Else:
7       Add [(#) the previous matched sentence] to FS
8       Add [(+) adding subsequence] from ESeq to FS
9     End
10  Return FS
```

Table II
ALGORITHM 2: FEATURE CHECKING

```
Input:  the standardized log sequence(LS) and the log feature database(FDB)
Output: the potential misconfiguration list
1   For every feature set in FDB:
2     FullScore := the number of (+) and (-) sentences
3     Score := the number of (-) sentences
4     For every feature in the feature set:
5       Find the matched (#)sentence in LS
6       If the feature is (+)feature:
7         Score += the max number of matched (+)sentences in order
8       If the feature is (-)feature:
9         Score -= the max number of matched (-)sentences in order
10    Similarity = Score / FullScore
11  Return the sorted Similarity List
```

E. Feature Checker

When the software fails, the diagnosing component uses the exception logs to troubleshoot configuration errors. We first preprocess the exception log sentences as mentioned above. Then, the feature checker will take every feature set from the log-feature database and calculate the similarity between the preprocessed log sequence and the feature set. That is, the checker calculates what percentage of the feature set is contained in the exception log sequence. If the log sequence contains some features, the target software environment may include misconfigurations.

Algorithm 2 shows the process undertaken by the feature checker. Its input is the preprocessed exception log sequence and the log-feature database. The checker calculates the similarity of one feature-set by adding the number of matched (+)sentences and subtracting the number of matched (-)

Table III
SOFTWARE AND CONFIGURATION INFORMATION

Software	Version	Option	Single-Misconfigurations	Multi-Misconfigurations
Httpd	2.4.6	564	7102	31
MySQL	5.5	671	8355	36
PostgreSQL	9.3.1	273	3451	22

Table IV
DIAGNOSIS RESULTS FROM MISCONFDOCTOR AND BOW

Software	**Misconfigurations**	**Software Fails**	**MisconfDoctor**	**BoW**
Httpd	100	93	78	56
MySQL	100	87	72	55
PostgreSQL	100	92	81	63
Sum	300	272	231	174
Ratio	–	100%	85%	64%

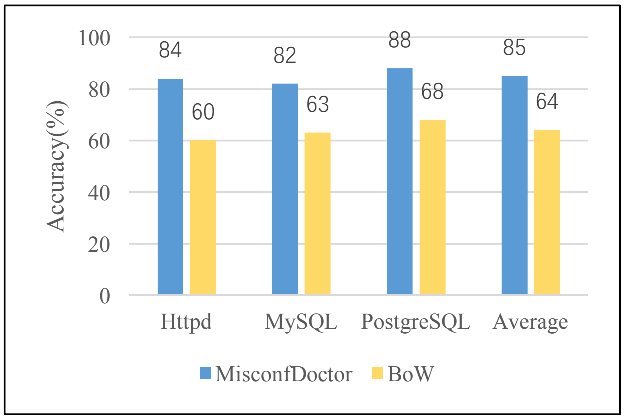


Figure 10. Accuracy Comparison between MisconfDoctor and BoW

sentences, which is a negative adjustment for similarity. Where there are two adjacent sentences in one feature, we set the distance to 5 while matching the new exception log, because some of the sentences in the new exception log may contain noise. Then the checker sorts the List [Misconfiguration, Similarity] by similarity. Finally, the user will be provided with the potential misconfigurations list.

V. EVALUATION

In this section, we will evaluate how effectively our method diagnoses configuration errors. Our experimental environment is a computer with an Intel Core(TM) i5-4590 CPU with 3.30GHz and 8G memory, and the operating system is Ubuntu 16.04.

We selected three mature and widely used pieces of open-source software (Httpd, MySQL and PostgreSQL) for our experiment, choosing a stable version of each type of software. Table 3 details the software studied and the configuration information. We extracted both single-configuration constraints and multi-configuration constraints and generated the misconfigurations (as shown in Table 3).

As mentioned in Section 4.2, we ran the target software's own test infrastructure and recorded the resulting exception logs and the corresponding configuration errors. Then, we extracted the exception log features from the reported log sequences.

A. Comparison Algorithm

In order to evaluate our method of extracting the features, we implemented a comparison algorithm in order to study the characteristics of the extracted features. Instead of checking the new exception log sequence using the log feature, as proposed in Section 2, the comparison algorithm used the bag-of-words (BoW) model to diagnose the configuration errors.

In information retrieval, the bag-of-words model assumes that (ignoring word order, grammar and syntax) a text is only a combination of words. The occurrence of each word in the text is independent and does not depend on whether other words appear. By using this method, we transformed the new exception log sequence into a word set after log preprocessing and we called the word set from the new exception log sequence as NES. Similarly, we transformed the log sequences with known misconfigurations into many word sets, which were called KES. We then represented each set as a word vector, where every element is 0 or 1, standing for whether the word exists or not. Then, we calculated the cosine similarity distance between the NES vector and the KES vectors.

For three of the types of software, we chose 100 misconfigurations from each in order to evaluate the accuracy of our method. The chosen misconfigurations include both single- and multi-option configuration errors. We also applied a stratified random sampling method to the options (according to their types) in order to make sure all the types of options had been covered. We then wrote a new test script for each piece of software by changing some operations in the ordinary test infrastructure. We checked the system's behavior using the new test suite, instead of directly using the software's own test infrastructure.

The results can be found in Table 4. The chosen 300 misconfigurations resulted in 272 cases of software failure, including failure to launch and failure to pass function tests. Some misconfigurations did not make the software fail, because the configuration error had been fixed by the program's robustness. For example, in PostgreSQL, we set the option "enable_sort" to "True", with the default value "on". The program corrected the value while reading the options.

We compared the accuracy of MisconfDoctor and BoW with the first option they recommended. Accuracy can be

Table V
Diagnosis results of Real-world Misconfigurations

Software	Misconfigurations	Diagnosis result	Ratio
Httpd	13	10	76%
MySQL	15	12	80%
PostgreSQL	12	9	75%
Sum	40	31	78%

interpreted as the portion of correctly diagnosed logs out of all the diagnosed logs. We can see, in Table 4, that MisconfDoctor diagnoses 85% of the failure cases, while BoW can only diagnose 64%. Fig. 10 shows the accuracy comparison applied to the three pieces of software, and again MisconfDoctor works better. The reason for this is that BoW ignores the sequence characteristics of logs, and so the word vector is affected by the normal log. As a result, the important log sentences and words decrease in effectiveness. MisconfDoctor fails to troubleshoot 15% of the failure cases because when the software handles the same type of configuration errors belonging to different options, the resulting logs may be the same or similar.

B. Real-world Misconfigurations

We searched StackOverflow and ServerFault to obtain real-world configuration problems that trouble users in the three target pieces of software. We choose 40 user cases of misconfigurations that had been caused by errors in the configuration files. All of the misconfigurations are Ubuntu-related and are provided along with the exception log by the users. The misconfigurations cover many aspects, including value ranges, data types and multi-option constraints. Data types include Boolean-type errors, path-type errors and enum-type errors. Table 5 lists both the configuration errors from each piece of software and the results returned by MisconfDoctor. We can see that MisconfDoctor diagnoses 78% of the errors effectively.

Due to the length limitations of this study, we cannot list each misconfiguration in detail. Instead, we have selected two typical cases to examine.

In the case from SstackOverflow, whose title is "Can't start MySQL server for the first time", the user installed the MySQL Server 5.5 Community Edition and ran the MySQL Server for the first time. The software would not start and the user was given the following log: "[ERROR] InnoDB: .ibdata1 can't be opened in read-write mode. [ERROR] InnoDB: The system tablespace must be writable" (Please note: this is only part of the long log.)

In response, MisconfDoctor diagnoses the exception log and concludes that "innodb_log_group_home_dir with access right error" is the first rank of the root cause with the similarity ranked at 0.83. The error is caused by the fact that permission to access the directory path innodb_log_group_home_dir has not been granted. The option

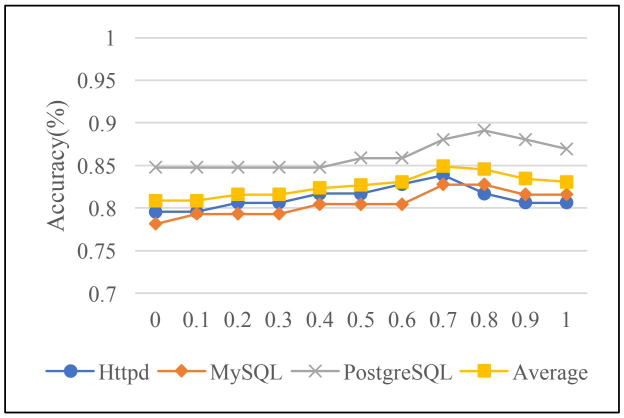


Figure 11. Accuracy with varied threshold

directs the directory path to the InnoDB log files; the server must have access rights in any directory where it needs to create log files. The user tried changing their data location by modifying the option, thereby solving the problem. The misconfiguration was one type of semantic misconfiguration and MisconfDoctor is effective at diagnosing the root cause. Other cases of successful diagnosis include "port with occupied error", "max_allowed_packet with value range error", "max_wal_senders and max_connections with value constraint error" and so on.

MisconfDoctor requires that the software to be diagnosed and the software used to extract the features are the same version. That is because an evolutionary version may treat options differently or may even remove some options. In a case where the user updated MySQL from version 5.5 to 5.7.9, MisconfDoctor made mistakes diagnosing the misconfiguration. After the upgrade, the user received the following error information: "[ERROR] unknown variable 'thread_concurrency=4' [ERROR] Aborting." The root cause is that, in MySQL 5.7, the variable thread_concurrency is removed and any reference to it will cause MySQL to not start.

In addition to the impact of software evolution, MisconfDoctor also failed to diagnose a problem caused by cross-stack configuration errors and some other complex operations we missed during testing. Overall, however, MisconfDoctor can effectively diagnose real-world configuration errors for a stable version of a piece of software within our test coverage.

C. Influence of Parameters

When filtering noise, we need the threshold to decide whether a sentence remains or not. If a log sentence occurs more often than the number of times specified by the threshold value, we regard it as a normal log sentence. The influence of the threshold is investigated in this section. We evaluated the relationship between MisconfDoctor's accuracy for the three target pieces of software with respect to different threshold values. We varied the threshold from 0 to 1 with a step of 0.1 each time. We evaluated the accuracy for each threshold with the same experiment as that used in Section 5.1. Fig. 11 shows that, when the threshold is less than 0.7, the average accuracy shows a gradual upward trend and that it gradually stabilizes at around 0.7. The reason for this is because, when the threshold equals 0, we ignore the noise log. This is sometimes the "warning" log that has been affected by the execution environment. When the threshold equals 1, we filter some important logs, which could affect the validity of the filtering feature. Upon consideration, we set the threshold to 0.7 in our experiment.

D. Discussion and Limitation

1) Mature logging system: Our approach is to diagnose configuration errors by using the software's exception log sequence features as signatures, which requires that the software should have ample logs that record the state of the system. In our experiments, the software we used (MySQL, Httpd, etc.) are all relatively mature software systems with detailed log systems. We also studied some software still under development with poor diagnostic results, because the logging systems were poor and the log information was sparse and vague in the event of a system failure. As such, we are calling on software developers to refine their logging systems so that they produce more detailed logs.

2) Comprehensive misconfigurations: Our tool can diagnose a variety of possible configuration errors in real life, which requires it to have the ability to generate comprehensive misconfigurations. Although the misconfigurations that we have generated are already very complex, we cannot guarantee that we will not miss some (particularly certain multi-option) misconfigurations.

3) High-coverage software test suits: We use each piece of software's official test infrastructure in order to test the software that has been injected with known misconfigurations. The aim is to expose the software's reaction to the error and to obtain exception log sequences. We assume that the software test coverage is high enough to fully simulate the user's real-life operation of the software. However, it is difficult to cover all the operations that users may conduct in their daily lives. In addition, where a piece of software does not have an official test module, the user needs to write the test suites by themselves. As a result, we are calling on software developers to provide mature and comprehensive test suites.

4) Cross-stack configurations and software evolution: Our tools require that the software used to learn the knowledge base should be the same version as the software that is to be diagnosed. This is because the placement and description of logs may change for evolved versions of the software, which may affect the accuracy of our results.

In addition, we did not consider the effect of cross-stack configuration errors, as discussed in Section 5.2. This will be one of our future research objectives.

VI. Related Work

A. Misconfiguration Detection and Diagnosis

The extensive research into misconfiguration detection and diagnosis mainly contains program analysis and checkpoint- and statistics-based approaches.

Program analysis methods [7-10] analyze both the corresponding statements given by configuration variables and the execution paths in order to infer the configuration options that caused the software system error. ConfAnalyzer [7] builds the mapping between configuration items and potential outliers in code and locates the fault's configuration option by retrieving the corresponding mapping. ConfAid [8] records and explores the impact of configuration value changes in the program traces in order to diagnose configuration faults. SherLog [9] uses runtime log messages to analyze and infer the execution path in order to diagnose failures. ConfDebugger [10] uses thin-slicing techniques to obtain configuration-read and error-stack-related statements in order to diagnose configuration errors. Such methods have defects in terms of their efficiency and scalability due to the path-explosion problem and because it is hard to go across the software boundaries.

Checkpoint-based approaches record the history of the system's states in a time series and check how the system transitions from the working state to the failure state when a system failure occurs. Chronus [34] uses a virtual machine and a time-travel disk to obtain historical snapshots of the system in order to find the changes to the configuration settings that caused the system to fail. Snitch [35] leverages the flight data recorder to construct the timeline views of the system states in order to troubleshoot the root cause.

Statistics-based studies can be divided into those that cover rule-based diagnosis [11-14] and those that cover signature-based diagnosis [29-32]. The rule-based methods learn rules from the correctly functioning system and its environment. If a configuration setting does not satisfy these learned rules, it will be labeled as a misconfiguration. Bauer et al. [12] apply association rule mining techniques to detect misconfigurations within access control policies. EnCore [13] enriches the plain configuration values using the environment information and unveils their implicit rules and correlations. CODE [14] uses a trie to represent the sequence of events and exploits the registry's access rules in Windows, detecting configuration errors by finding violations of the rules. Such rule-based approaches are prone to false positives because it is difficult to guarantee the completeness of the rules and avoid learning inaccurate rules.

Signature-based diagnosis approaches attempt to reason out the configuration errors by comparing the signature of the failed execution with the reference signatures. Yuan et al. [29] describe the system signature according to the system call execution sequence for the historical failures. Strider [32] takes the contents of the registry as the state of the systems and uses the state-differencing method to narrow down the range of misconfigurations. AutoBash [30] uses the success and failure patterns of known predicates to diagnose configuration errors. SigConf [31] describes system signatures and features of the known configuration errors based on coarse-grained object-access sequences (such as files, class libraries, etc). However, these approaches to misconfigurations aren't comprehensive, so they are limited by the size and comprehensiveness of the reference signatures, which greatly effects their accuracy. MisconfDoctor, based on comprehensive misconfigurations, describes the system features according to the exception log features. As such, it is much more effective.

B. *Configuration Testing*

The generation of comprehensive miconfigurations is a prerequisite needed for feature extraction and diagnostics in our work. Configuration testing contributes to our work. ConfErr [17] is a pioneer in configuration testing, even though the alternation rules (e.g., omissions, substitutions and case alternations of characters) it uses to generate misconfigurations are simple. SPEX [18] extracts constraints through a program analysis of the source code and generates constraint-based misconfigurations in order to evaluate the software's response ability. However, its coarse-grained constraints do not meet our high-coverage requirements. In our work, based on our classification in ConTest [21] and our program analysis, we can extract fine-grained option constraints in order to generate misconfigurations, which provides higher coverage and is comprehensive.

VII. Conclusion

Misconfiguration has become one of the major causes of software failure. In this paper, we propose a method with which to diagnose misconfigurations via log-based configuration testing. We implement a tool named MisconfDoctor that can generate comprehensive misconfigurations and obtain an exception-log feature database. Through comprehensive misconfiguration testing, MisconfDoctor first extracts the log features for every misconfiguration and builds a feature database. When system misconfiguration occurs, MisconfDoctor automatically suggests potential misconfigurations by calculating the similarity of the new exception log to the feature database. We use manual and real-world error cases from Httpd, MySQL and PostgreSQL in order to evaluate the effectiveness of out method. Experimental results demonstrate that MisconfDoctor's accuracy reaches 85% in manual-error cases and 78% in real-world cases.

Acknowledgment

The work described in this paper was supported by National Natural Science Foundation of China (Project No.61690203 and U1711261); National Key R&D Program of China (Project No.2017YFB1001802 and No. 2017YFB0202201).

References

[1] Barroso, L, J. Clidaras, and U. Hoelzle. "The Datacenter as a Computer:An Introduction to the Design of Warehouse-Scale Machines." Morgan & Claypool, 2009:154.

[2] Rabkin, Ariel, and R. H. Katz. "How Hadoop Clusters Break."IEEE Software30.4(2013):88-94.

[3] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in Conference on Usenix Symposium on Internet Technologies and Systems, 2003, pp. 1-1.

[4] Yin, Z., Ma, X., Zheng, J., Pasupathy, S., Pasupathy, S., & Pasupathy, S. (2011). An empirical study on configuration errors in commercial and open source systems.ACM Symposium on Operating Systems Principles(pp.159-172). ACM.

[5] Oppenheimer, David, A. Ganapathi, and D. A. Patterson. "Why Do Internet Services Fail, and What Can Be Done About It?"Usenix Symposium on Internet Technologies and Systems2003:165-171, in press.

[6] R. Johnson. More Details on Today's Outage. http://www.facebook.com/note.php?note id=431441338919, 2010.

[7] Rabkin A, Katz R. Precomputing possible configuration error diagnoses [C]. In Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on. 2011: 193-202.

[8] Attariyan, Mona, and J. Flinn. "Automating configuration troubleshooting with dynamic information flow analysis."Usenix Conference on Operating Systems Design and ImplementationUSENIX Association, 2010:237-250.

[9] Yuan, D., Mai, H., Xiong, W., Tan, L., Pasupathy, S., & Pasupathy, S. (2010). SherLog: error diagnosis by connecting clues from run-time logs.Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems(Vol.38, pp.143-154). ACM.

[10] Zhang, Sai. "ConfDiagnoser: an automated configuration error diagnosis tool for Java software." International Conference on Software Engineering IEEE, 2013:1438-1440.

[11] Palatin, N., Leizarowitz, A., Schuster, A., & Wolff, R. (2006). Mining for misconfigured machines in grid systems.Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, Pa, Usa, August(pp.687-692). DBLP.

[12] Bauer, L., Garriss, S., & Reiter, M. K. "Detecting and resolving policy misconfigurations in access-control systems."Acm Transactions on Information & System Security14.1(2011):1-28.

[13] Zhang, Jiaqi, et al. "EnCore: exploiting system environment and correlation information for misconfiguration detection."Acm Sigarch Computer Architecture News49.1(2014):687-700.

[14] Yuan, D., Xie, Y., Panigrahy, R., Yang, J., Verbowski, C., & Kumar, A. (2011). Context-based online configuration-error detection.Usenix Conference on Usenix Technical Conference(pp.28-28). USENIX Association.

[15] Jiang, W., Li, Z., Li, Z., Li, Z., Li, Z., & Zhou, Y. (2009). Understanding customer problem troubleshooting from storage system logs.Proccedings of the, Conference on File and Storage Technologies(pp.43-56). USENIX Association.

[16] W. Jiang. Understanding storage system problems and diagnosing them through log analysis. Ph.D. Dissertation. (2009).

[17] Keller, L, P. Upadhyaya, and G. Candea. "ConfErr: A tool for assessing resilience to human configuration errors."IEEE International Conference on Dependable Systems and Networks with Ftcs and DCCIEEE, 2008:157-166.

[18] Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., & Yuan, D., et al. (2013). Do not blame users for misconfigurations.Twenty-Fourth ACM Symposium on Operating Systems Principles(pp.244-259). ACM.

[19] StackOverflow, https://stackoverflow.com/

[20] ServerFault, https://serverfault.com/

[21] Wang Li, Shanshan Li, Xiangke Liao, Xiangyang Xu, Shulin Zhou, and Zhouyang Jia. 2017. ConfTest: Generating Comprehensive Miscon guration for System Reaction Ability Evaluation . In Proceedings of EASE'17, Karlskrona, Sweden, June 15-16, 2017, 10 pages.

[22] Shulin Zhou. Evolution-Oriented Failure Diagnosis of Software Configuration: [D].Changsha: National University of Defense Technology, 2016. [in Chinese].

[23] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the 2004 Inter- national Symposium on Code Generation and Op- timization (CGO'04), March 2004.

[24] The Apache HTTP Test Project. Retrieved March 20, 2018 from http://httpd.apache.org/test/

[25] The MySQL Test Framework. Retrieved March 20, 2018 from https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN.html

[26] PostgreSQL 9.3.23 Documentation. Retrieved March 20, 2018 from https://www.postgresql.org/docs/9.3/static/regress-run.html

[27] M. Porter. An algorithm for suffix stripping. Program, 14(3):130-137, 1980.

[28] Dan, Gusfield.Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, 1997.

[29] Yuan, C., Lao, N., Wen, J. R., Li, J., Zhang, Z., & Wang, Y. M., et al. (2006). Automated known problem diagnosis with event traces.Acm Sigops Operating Systems Review,40(4), 375-388.

[30] Su, Ya Yunn, M. Attariyan, and J. Flinn. "AutoBash:improving configuration management with operating system causality analysis." ACM, 2007:237-250.

[31] Attariyan, Mona, and J. Flinn. "Using Causality to Diagnose Configuration Bugs. "Usenix Technical Conference, Boston, Ma, Usa, June 22-27, 2008. ProceedingsDBLP, 2008:281-286.

[32] Wang,Y.M.,Verbowski,C.,Dunagan,J.,Chen,Y.,Wang,H.J.,&Yuan, C., et al. (2003). Strider: a black-box, state-based approach to change and configuration management and support. Science of Computer Programming, 53(2), 143-164.

[33] Xu, Tianyin, and Y. Zhou. "Systems Approaches to Tackling Configuration Errors: A Survey."Acm Computing Surveys47.4(2015):1-41.

[34] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. 2004. Configuration debugging as search: Finding the needle in the haystack. In Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04).

[35] James Mickens, Martin Szummer, and Dushyanth Narayanan. 2007. Snitch: Interactive decision trees for troubleshooting misconfigurations. In Proceedings of the 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SYSML'07).

[36] Yuan D, Park S, Huang P, Liu Y, Lee MM, Tang X, Zhou Y, Savage S. Be conservative: enhancing failure diagnosis with proactive logging. In: Proc. of the 10th Symp. on Operating Systems Design and Implementation (OSDI). 2012. 293-306.

[37] Jia, Zhouyang, et al. "SMARTLOG: Place error log statement by deep understanding of log intention." IEEE, International Conference on Software Analysis, Evolution and Reengineering IEEE Computer Society, 2018:61-71.