Do You Really Know How to Configure Your Software? Configuration Constraints in Source Code May Help

Xiangke Liao, Shulin Zhou, Shanshan Li, Zhouyang Jia, Xiaodong Liu, and Haochen He

***Abstract*—Misconfigurations have become one of the major causes of software failures because of their increasing prevalence and severity. The complexity of configurations and users' lack of domain knowledge are the main reasons for massive misconfigurations. Users usually identify and diagnose misconfigurations by making a comparison against the conditions that configuration options should satisfy, which we refer to as *configuration constraints*; however, sometimes it is hard for users to accomplish this work. Some work has been done on obtaining configuration constraints, especially from source code; nevertheless, only part of the situation has been considered, such as if-statement code snippets, limiting its help in misconfiguration diagnosis. In order to better extract configuration constraints for users' guidance and misconfiguration diagnosis, we carried out a comprehensive manual study on the existence and variance of the configuration constraints in the source code of five different pieces of widely used open-source software. Three categories of findings are summarized based on our study, namely the general statistics, the general features of specific kinds of constraints, and the obstacles to the automatic extraction of configuration constraints. With these findings, we proposed several suggestions to maximize the automatic extraction of configuration constraints. The results show that our suggestions could improve the extraction of configuration constraints compared to existing methods.**

***Index Terms*—Automatic extraction, misconfiguration, misconfiguration diagnosis, configuration constraints, software reliability.**

I. Introduction

IN RECENT years, software misconfigurations have drawn tremendous attention for their increasing prevalence and severity. Yin *et al.* [1] pointed out that misconfigurations accounted for 27% of failure cases in a major data storage company in the U.S. Also, misconfigurations were the second biggest cause of service outages in Google's main services, accounting for nearly 28%, as stated in the report by Barroso *et al.* [2]. Furthermore, Rabkin and Katz [3] mentioned that misconfiguration is the leading cause of Hadoop cluster failures and the dominant source of support costs, considering the number of failure cases as well as the total technique support time. Many other works [4]–[6] also pay attention to such misconfiguration problems, and many studies [7]–[10] have tried to address it from different points of view.

```
<VirtualHost 192.168.1.11:80>
    ServerName www.test1.com
    DocumentRoot /www/test1/
    <Directory "/www/test1">
        Options Indexes FollowSymLinks
        AllowOverride None
        Order allow,deny
        Allow From All
    </Directory>
</VirtualHost>
```

Fig. 1. Example of VirtualHost setting in Apache Httpd Server.

The reasons for misconfigurations are mainly twofold. On the one hand, users often lack domain knowledge for the software and its configuration options, limiting their correct usage of configuration options. For instance, for better flexibility and scalability, software developers usually provide hundreds of configuration options for current database [11] and web servers [12], which presents a great challenge for novices and even experienced users in setting them correctly. Xu *et al.* [13] also indicate that, "first, a significant percentage (up to 48.5%) of configuration issues are about users' difficulties in finding or setting the parameters to obtain the intended system behavior; second, a significant percentage (up to 53.3%) of configuration errors are introduced due to users' staying with default values incorrectly," which gives data to support our opinion. On the other hand, the complexity in the configuration procedure also restricts the usage of software, as mentioned in [13]. In detail, every single configuration options value should satisfy a valid range; for instance, the port in Redis should be an integer in the range from 0 to 65535. Some structural modules should be well formatted; for example, in Apache Httpd Server, the setting for virtual host should be between the terms "<VirtualHost>" and "</VirtualHost>," as shown in Fig. 1. In addition, there can be complex relationships between different configuration options. Take PostgreSQL as an example; the value of configuration

Manuscript received July 31, 2017; revised January 9, 2018 and March 16, 2018; accepted April 28, 2018. Date of publication June 15, 2018; date of current version August 30, 2018. This work was supported in part by the National Natural Science Foundation of China under Grant 61690203 and Grant 61532007, and in part by the National 973 Program of China under Grant 2014CB340703. This paper was presented in part at the 21st International Conference on Evaluation and Assessment in Software Engineering, Karlskrona, Sweden, June 2017 [27]. Associate Editor: J. Zhang. *(Corresponding author: Shulin Zhou.)*

X. Liao is with the College of Software, Tsinghua University, Beijing 100084, China, and also with the College of Computer, National University of Defense Technology, Changsha 410073, China.

S. Zhou, S. Li, Z. Jia, X. Liu, and H. He are with the College of Computer, National University of Defense Technology, Changsha 410073, China.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2018.2834419

option "superuser_reserved_connections" must be less than "max_connections," otherwise the PostgreSQL daemon will close down. Furthermore, if the value of configuration option "client_request_buffer_max_size" is less than the value of configuration option "request_header_max_size," the client-request package might overflow without enough buffer to hold the header of the request, let alone the request's body.

The typical routine for misconfiguration diagnosis is to check whether the current configuration settings satisfy a certain condition when a misconfiguration happens. We refer to this certain condition hereinafter as a *configuration constraint*. If these configuration constraints could be illustrated in advance, there would be many benefits for achieving the correct configuration of the software system, as well as misconfiguration diagnosis and repair.

As important references for using software, user manuals and official documentation may have information about configuration constraints in natural language or formatted structures, but it is hard for users to find them in thousand-page documents. On the other hand, some developers may not maintain well-formatted documentation, much less the information of configurations that users need. More seriously, if the software documentation is not updated in a timely manner, the constraints recorded in the documentation could be totally wrong, thus increasing the difficulty of misconfiguration diagnosis. Therefore, some other approaches are needed to provide the configuration constraints for users.

To obtain the configuration constraints, current literature has made much effort. Based on summarizations of three commonly used mapping structures, SPEX [14] extracts configuration constraints from source code based on predefined patterns, such as if-statement condition checking and handling, lib-function calling trace, and so on. Then, based on those configuration constraints, they injected some faults to expose the vulnerabilities in the software and detect any inconsistencies between configuration options. In addition, they also propose several suggestions about configuration design and implementation. The same effort is also made in [15]. Focusing on the key-value pair model in configuration files, they present a static analysis technique to acquire configuration options from source code with a few manual labeling efforts. In order to reduce the burden of maintaining configuration documentation, they also gave some effort to inferring the basic type of configuration options. EnCore [16] tries to learn configuration constraints from thousands of users' configuration files by machine learning algorithms. It is innovative that EnCore could extract potential and complicated constraints between different configuration options or even environment variables. Nevertheless, the requirement for predefined constraint templates, which reduces the search space in machine learning, limits the applicability to common users.

Considering the urgent demand for obtaining configuration options for users, as well as our conjecture that there must be some checking mechanism for configuration, we hold the opinion that it is easy to obtain configuration constraints from source code with explicit forms. However, we found it is often not the case based on our study on real-world software. We manually studied five pieces of widely used open-source

```
/* postgresql-9.5.6/src/backend/postmaster/postmaster.c */
...
 if (ReservedBackends >= MaxConnections)
 {
  write_stderr("%s: superuser_reserved_connections must be
      less than max_connections\n", progname);
  ExitPostmaster(1);
 }
...
```

Fig. 2. Example of configuration constraint in if-statement-checking code snippets in PostgreSQL.

```
/* postgresql-9.5.6/src/backend/utils/misc/guc.c */
...
 {
  {"geqo_effort", PGC_USERSET, QUERY_TUNING_GEQO,
   gettext_noop("GEQO: effort is used to set the default
       for other GEQO parameters."),
   NULL
  },
  &Geqo_effort,
  DEFAULT_GEQO_EFFORT, MIN_GEQO_EFFORT, MAX_GEQO_EFFORT,
  NULL, NULL, NULL
 },
...
```

Fig. 3. Example of value range constraints in PostgreSQL.

software, and found that the forms of configuration constraints vary in different kinds of software, which are not always in simple if-statement checking situations. For instance, in PostgreSQL, there might be some constraints in if-statement checking, as shown in Fig. 2, but developers also use some other forms, such as specific structures, to implement the value range restrictions. As shown in Fig. 3, the value range of configuration option "GEQO_EFFORT" is defined by structure members "MIN_GEQO_EFFORT" and "MAX_GEQO_EFFORT," i.e., from 1 to 10. Then, sometimes there is much semantic information in the context-sensitive configuration constraints in specific kinds of software, leading to the rare possibility to achieve automatic extraction using simple program analysis. Furthermore, as a consequence of bad configuration design or developers' negligence, there might be no checking mechanism regarding configuration constraints at all.

Faced with such situations, we carried out a comprehensive manual study to uncover the characteristics of configuration constraints in source code. On that basis, we would be able to fulfill the automatic extraction of configuration constraints with respect to the prevention and diagnosis of misconfiguration. In detail, we manually studied five typical widely used open-source software packages at first. In light of the overall results, three categories of findings are summarized from different aspects, namely the general statistics, the general features of specific kinds of constraints, and the obstacles to the automatic extraction of configuration constraints. The general statistics consist of the proportion of configuration constraints that can be extracted from source code, the proportion of configuration constraints for configuration options with different basic types, and the general existing forms of configuration variables in source code. The general features of specific kinds of constraints mainly focus on the various forms of configuration constraints, the clustered definition and declarations about enumeration constraints, and the semantic information that could be used in constraint

extraction. Finally, the obstacles to extracting configuration constraints mainly lie in the structural configuration options and the resource-related configuration options.

Based on the aforementioned findings, we propose three suggestions about automatic extraction for different kinds of configuration constraints. First, for the configuration options that use uniform mapping structures, we extracted the numeric value range of these configuration options with statistical analysis as well as some semantic information. Then, considering the clustered code snippets for enumeration, some text analysis was used to extract the value space of the enumerative configuration options. Finally, with some semantic information on the identifiers of the function parameters, we improved the dataflow analysis to obtain semantic meanings of the configuration options. The experiment's results show that our methods are efficient in extracting configuration options using these suggestions.

The rest of this paper is organized as follows. We introduce the methodology in Section II. Section III describes in detail the findings regarding configuration constraints in source code in detail. The strategies for the automatic extraction of configuration constraints and the corresponding experiments are illustrated in Section IV. Some related work is introduced in Section V. Finally, we present our conclusion in Section VI.

II. Methodology

In this section, we will mainly introduce our methodology for investigating the existence and variance of configuration constraints in source code. First, based on the research of Xu *et al.* [17], as well as our survey, we hold the view that, presently, there are no mature tools or approaches for extracting configuration constraints from source code, except for some trials, namely SPEX [14] and Rabkin and Katz [15]. Therefore, we tried to retrieve configuration constraints manually from source code to figure out the characteristics.

In detail, first of all, we built the collection of configuration options for every studied software packages by searching the official documentation and manuals. In this procedure, we first search the Internet to collect commonly used configuration options for every software; then, we search the occurrence of these commonly used configuration options in documentation to locate the most possible chapters or sections about the description of configuration options, and collect all the configuration options occur; finally, based on the chapter name and meaning, we search catalog of documentation to find other possible configuration options.

Then, we built the mapping relationship between the configuration options in the configuration file (or our collection of configuration options) and the program variables in the source code of the target software. Based on the hypothesis that configuration options perform as controllers at runtime as well as our manual statistics from the survey (mainly in Finding 3 of Section III), we believe that configuration variables are usually used directly without complex assignment and propagation. Therefore, we manually checked every use scenario for the configuration variables to obtain the relevant configuration constraints.

TABLE I
Basic Information on the Software Studied

Software	Description	Version	LOC
MySQL	Database Server	5.7.16	1.2M
Apache Httpd	Web Server	2.4.23	148K
Redis	In-Memory Database	3.2.5	112K
Postfix	Mail Transfer Agent	3.1.3	230K
PostgreSQL	Database Server	9.5.6	1.1M

TABLE II
Explanation of Configuration Types

Configuration Type		Explanation
Simple Type	Numeric Type	Includes integer and float types.
	Enumeration Type	Include enumerations, Boolean, and some macros used as enumeration.
	String Type	Other common strings, such as file path, user name, etc.
Complex Type		Structural configuration options and software-specific encapsulated configuration options.

The categories of configuration constraints we studied will be introduced in following section.

A. Dataset

To cover all possibilities as far as we can, we choose MySQL, Apache Httpd, Redis, Postfix, and PostgreSQL as our target software packages. These five pieces of open-source software are all widely used throughout the world with a long developing history. To ensure the reliability and effectiveness of our study, we took the newest stable version of each of these five software packages into the study. The basic information about these software packages is listed in Table I.

B. Study Methodology

In order to figure out the existence of configuration constraints in source code, we first determined the main types of configuration options. Based on the configuration type classification in ConfTest [18], as well as our survey on their usage in official documentation and source code, we found that configuration options may be of various kinds of types and forms, but their basic type can be classified into limited categories. Considering the actual usage and generality of the configuration options in different software packages, we classify the configuration types as simple type and complex type, where the simple type consists of numeric, string, and enumeration, which are listed in Table II. In particular, we propose the complex type that covers all the structural configuration options and software-specific encapsulated configuration options that are difficult to present using the simple basic types. For instance, in Apache Httpd, the configuration of a virtual host needs the configuration option "<VirtualHost>" to be modularized, as shown in Fig. 1, and configuration option "Options" sets a series of attributes for a given directory, for instance, "Options Indexes

TABLE III
CONFIGURATION CONSTRAINTS AND THEIR EXPLANATIONS

Constraint Types		Explanations
Single-dimension Constraint	Value Range	The value range a numeric configuration option should satisfy.
	Enumeration	The value space an enumeration configuration option should satisfy.
	Semantic Meaning	The semantic meaning of the configuration option.
Multiple-dimension Constraint	Value Control	The usage of a configuration option relies on another configuration option.
	Multi-value Relationship	The value relationship two or more different configuration options should satisfy.

FollowSymLinks." It is difficult to extract constraints from those configuration options using general analysis methods, so we classify them as configurations with complex types. To be clear, our definitions of the basic types of configuration options in Table II are more coarse-grained than those in ConfTest [18]. That is because some types of configuration options only exist in a few software packages, and some of the code snippets that handle these types are complex and unique, so it is difficult and unnecessary to summarize a pattern for those configuration options in our view. Therefore, in our study, we roughly classify all the configuration options into the aforementioned four types.

With respect to configuration constraints, ConfTest [18] defines specific syntactic and semantic constraints for different types of configuration options, and SPEX [14] defines several patterns of code formats as their configuration constraints. Considering the fact that source code should contain all the information about the processing procedure of software, including handling of configurations, so we could analyze source code to extract configuration constraints. Based on the summaries of the code patterns for the configuration constraints of configuration options with different basic types, as well as the constraints referenced by ConfTest [18] and SPEX [14], we define our own constraint categories in Table III. In general, our constraint categories cover most of the possible situations for the simple types, and those constraint types represent the main forms of configuration constraints. Specifically, the three constraint types in the single-dimension constraint category correspond to three simple configuration types, representing the possible constraints we could extract from source code for different basic configuration types. Considering the fact that complex types have various code formats, we can hardly ever summarize a general code pattern for the constraints of the structural configuration options and software-specific encapsulated configuration options in different software packages. Therefore, the constraints of the complex type are not analyzed in our study (There are still methods for implementing the constraint extraction of some configuration options in the complex type, but the diversity of configuration options in different software packages makes the method less generalizable. Hence, in our study, we omit the constraints of configuration options with complex types).

The single-dimension constraints contain the constraints that restrain one single configuration option, while multiple-dimension constraints restrain more than one configuration option. In our study, most of these constraints consist of two configuration options or more, so we name them multiple-dimension constraint. In detail, the single-dimension constraints consist of the semantic meaning, value range, and enumeration, and the multiple-dimension constraints consist of the value control and multivalue relationship. The explanation of these configuration constraints are as follows.

The semantic meaning of a configuration option represents the common resource type that the configuration option represents. In this paper, we mainly consider five types of semantic meaning, namely FILE, DIR, PORT, IP, and URL. For instance, Apache Httpd has a configuration option "ServerRoot," whose value is the directory in which Apache Httpd is installed, so the semantic meaning of "ServerRoot" is DIR. As mentioned above, we can also determine other types of semantic meaning by summarize the specific patterns of handling code snippets relate to other resource types. But, after our study, we found that other types of semantic meaning is rare in our studied software packages. So, we only considered these five types of semantic meaning in our present study.

The value range of a configuration option limits the values that this configuration option can be set to. Generally, only numeric configuration options have constraints of value range. Taking the configuration option "deadlock_timeout" in PostgreSQL as an example, the value of "deadlock_timeout" should be an integer equal to or greater than 1.

The enumeration of a configuration option refers to the value space that the configuration option could be assigned to. In most of the studied software, there are configuration options that have several modes to adjust or control the program runtime trace, which has limited values for a string setting. If users use a string that is not in the value space, the program might be running in an undesired way. For example, the configuration option "log_timestamps" in MySQL could only be set as "UTC" and "SYSTEM," so the enumeration of "log_timestamps" is "{UTC, SYSTEM}."

The value control defines the relationship between different configuration options. In a value control relationship, whether a configuration option takes effect depends on the value of another configuration option; we call them the controlee and controller configuration options. For instance, in PostgreSQL, only when the configuration option "enable_GEQO" is set to true can the configuration options "GEQO_threshold" and "GEQO_pool_size" take effect. In our study, we only take Boolean configurations as the controller configurations. To support our conclusion, we randomly selected 20 different code snippets that satisfy the value-control pattern while the controllers are not Boolean configuration options. All of them are similar to the example illustrated in Fig. 4. The configuration options "log_min_messages" and "client_min_messages" are in the code pattern for value control, but, obviously, this is not a

```
/*postgresql-9.5.6/src/backend/access/transam/xlogutils.c*/
...
if (log_min_messages <= DEBUG2 || client_min_messages <=
    DEBUG2){
      char *path = relpathperm(hentry->key.node, forkno);
      elog(DEBUG2, "page %u of relation %s has been dropped"
          ,
           hentry->key.blkno, path);
      pfree(path);
}
...
```

Fig. 4. Example of a non-value-control constraint in the same code pattern in postgresql-9.5.6.

constraint for both of these enumerative configurations. Considering the aforementioned support data and the fact that there are frequent situations whereby different configuration values are used in one condition to control a different program's runtime routine in different situations, we hold the opinion that those code patterns that non-Boolean configuration options perform as controllers are not configuration constraints.

Finally, the multivalue relationship identifies the conditions that the value of two or more different configuration options should satisfy. In reality, different configuration options may have a potential relationship, especially between numeric configuration options; for instance, the configuration options "superuser_reserved_connections" and "max_connections" in PostgreSQL, as mentioned in Section I.

To be clear, the value control and the multivalue relationship constraints are both related to multiple configuration options, so they are similar to some degree. However, there are still some differences between them. The value control constraint emphasizes the dominance of the controller configuration option, and the value of the controlee option is almost ignored. While the configuration options in multivalue relationship constraints are equal to each other. This is the biggest difference between them. Therefore, we classify both these kinds of constraints as multiple-dimension constraints.

Based on those configuration types and constraint categories, we carried out a comprehensive manual study about the existence and variance of configuration constraints in source code. In detail, first, two master students and one doctoral student of our group studied the source code and summarized the possible code patterns of configuration constraints separately. Then, they compared and discussed the code patterns together, and determined the target code pattern for different kinds of configuration constraints. To be clear, we counted the number of configuration constraints using the following rules: if a configuration option has several constraints of one type (for example, configuration option "opt $>$ 0" and "opt $<$ 100"), then we count it as one constraint; if a configuration option has several constraints of different types (for example, configuration options "opt1 $>$ 0" and "opt1 $<$ opt2"), then we count it as two; if two configuration options form a multiple-dimension constraint, then we count it as two constraints, with one for each option. Next, every student completed the study for different software packages separately and recorded the constraints. To ensure the correctness and completeness of the study, they cross-checked the results of the manual extraction. We ensured that there were at least two students who manually analyzed every piece of software. When there was a disagreement between two students, they first rechecked the constraints in source code by themselves and tried to come to an agreement. If they could not reach an agreement, the third student got involved and gave a final decision about the results. All of these three students have at least two years of experience in software development. The overall study took almost ten man-months.

C. Threats to Validity and Limitations

Considering the limitations of manual analysis, our study is subject to a validity problem. Specifically, the main sources come from whether these five software packages could represent general situations and whether our defined constraint types cover most situations.

In order to ensure the representiveness of the target software, we chose these five software packages with diverse functionality, and all of them are ranked top in their own software categories. As for the representiveness of our defined constraints types, to ensure most situations were covered, we did a detailed survey to collect the common configuration types, then defined the format and types of our target configuration constraints. In our study, these constraint types cover the main forms of configuration constraints in source code. Although there are indeed some forms of configuration constraints that are not covered by our constraints set, it is barely possible to extract these constraints automatically, or even by manual analysis. Therefore, we could ensure that our defined constraint types represented the most common situations.

III. Findings in Real-World Source Code

Based on our study's results,[1] we summarized seven findings from different aspects. In this section, we will illustrate our findings in detail, mainly focusing on the existence statistics for configuration constraints, inspirations for extracting configuration constraints, and the challenges in the automatic extraction of configuration constraints.

A. General Statistics for Configuration Constraints

Finding 1: Based on the program analysis, we were able to extract 64% of the configuration constraints with various code forms on average, while current research only covers 27%, mainly in if-statement situations.

Table IV shows the existence of configuration constraints in different software. It is common sense that different software packages have different levels of configuration checking mechanisms. The data in the second column shows that Redis has the highest proportion of configuration constraints (91.7%), while Apache Httpd is the poorest one in terms of configuration checking (at 37.5%).

Faced with such situations, with further analysis, we find that the causes of these results are obvious. Redis has structured if statements to check the majority of its configuration options,

[1]All the configuration constraints of our manual study are available at https://github.com/zhou-shulin/configuration-constraints.

TABLE IV
DIFFERENT PROPORTIONS OF CONSTRAINTS IN DIFFERENT SOFTWARE

Software	Proportion of configuration constraints	Proportion of configuration constraints that SPEX could extract
PostgreSQL	76.0% (136[1]/ 179[2])	44.1% (60[3]/136[1])
Redis	91.7% (44/48)	84.1% (37/44)
MySQL	67.5% (197/292)	4.1% (8/197)
Httpd	37.5% (33/88)	39.4% (13/33)
Postfix	49.5% (36/91)	8.3% (3/36)

[1] The number of configuration options that could extract constraint types mentioned in Table III.
[2] The total numbers of configuration options we studied in the manuals and documents related to the software.
[3] The number of configuration options that could extract constraints by SPEX.

so the proportion of configuration constraints is very high. In Httpd, a series of set-functions is used to fulfill the mapping between the configuration value and the relevant variable, and some of the configuration options are of the string or complex type. The assignment procedures in some of these set-functions are too complex, limiting the extraction of the configuration constraints.

Moreover, with regard to the data in last column of Table IV, we can see the gaps between the existing constraints in the source code and the numbers that current research obtained. With further analysis, we reveal that the causes are the varying forms of configuration constraints in different software packages that current research does not consider. This will be illustrated in detail in Findings 3 and 4. For instance, in SPEX [14], the if statements in which the configuration variables are used as judging conditions are first located. Then, SPEX defines several handling measures to be the failure signals of the software, namely program exits, aborts, returns an error code, and resets the value of the configuration variable. When these handling measures are used in the branch block of the if statement, SPEX inverts the judging condition as a configuration constraint; otherwise, it takes the judging condition as a configuration constraint. Nevertheless, few configuration constraints are in this if-check form in our study results, leading to the lower proportion in last column of Table IV.

Finding 2: In terms of basic configuration types, numeric configuration options are better checked, while configuration options of the string type are always poorly checked.

Table V shows the constraint distribution of different configuration option types. It is obvious that, on average, for 98.6% of configuration options of the numeric types (i.e., integer and float), the relevant constraints could be retrieved, most reached 100%, except for Httpd. This makes sense because numeric configuration options are easy to check and prone to mistakes, so the checking mechanism is enough. As for string-type configurations, in view of the flexible format of string-type configurations, few types could be checked well, such as file path, directory, and IP. When it comes to configuration options of the complex type, it is barely possible to extract certain common configuration constraints, owing to the otherness in the forms and handling logic among different configuration options and software, as well as the rich semantic context. For these results, we hold the view that the proportion of configuration constraints with numeric and enumeration types should reach 100%, otherwise the checking is weak. As for the configuration options of the string and complex types, the proportion of configuration constraints should be as high as possible.

Finding 3: The forms of the variables in different software packages may be different, but they are mainly global variables or members of global structures.

In order to find the configuration constraints in the source code, first, we needed to know the forms of the configuration options in the source code (i.e., the relevant variables of the configuration options), which we refer to as configuration variables. Through our analysis, we determined that configuration variables are mainly global variables or members of global structures. Detailed information is listed in Table VI.

As Table VI shows, PostgreSQL and Postfix use global variables as their configuration variables. Redis uses a global structure object "server" to store all the configuration variables. In MySQL, global variables and structures are both used to store different kinds of configuration variables with different effect scopes. In Httpd, some structures are used to store different configuration variables.

Furthermore, the assignment of configuration variables is also different. PostgreSQL, MySQL, and Postfix mainly use structural modules to fulfill the assignment of configuration variables. Redis uses if-statement checking to distinguish different configuration options and assign the setting values to configuration variables. Httpd uses a series of set-functions to accomplish the task. To reach the goal of increasing code reuse and creating a uniform interface, Httpd and MySQL use the offset of the configuration variables in the structure to distinguish them when using the same function call. For instance, configuration options "ServerAdmin" and "ErrorLog" in Httpd use the same set-function "set_server_string_slot" to accomplish the assignment, as shown in Fig. 5.

Whilst most of these configuration variables are used in the original forms without reassignment, few are reassigned to local variables for the purpose of efficiency. In particular, none of these configuration variables are used in the form of offsets in structures. This finding is instructive in the search for configuration constraints in source code.

B. General Features of the Existence of Configuration Constraints

Finding 4: Configuration constraints have various forms, rather than if-condition statements.

In the survey before our study, we found that current research focuses on the if-checking patterns to extract configuration constraints from source code. This makes sense under certain circumstances. For instance, for the code snippet for the configuration option "port" in Redis-3.2.5, as shown in Fig. 6, it is easy to extract its constraint as "$0 \leq$ port ≤ 65535." Nevertheless, there are quite a few situations that do not fall into this pattern based on our study results, especially for configuration options of the numeric type. Hence, we need further approaches

TABLE V
CONSTRAINT PROPORTIONS FOR CONFIGURATION OPTIONS WITH DIFFERENT BASIC TYPES

Software	Numeric	Enumeration	String	Complex
PostgreSQL	100.0% (106/106)	100.0% (22/22)	13.7% (8/51)	-
Redis	100.0% (36/36)	100.0% (4/4)	50.0% (4/8)	-
MySQL	100.0% (170/170)	100.0% (24/24)	6.4% (3/47)	0.0% (0/51)
Httpd	64.3% (9/14)	100.0% (20/20)	28.5% (4/18)	0.0% (0/36)
Postfix	100.0% (31/31)	-	8.3% (5/60)	-

TABLE VI
FORMS OF CONFIGURATION VARIABLES IN DIFFERENT SOFTWARE PACKAGES

Software	Forms of configuration variables
PostgreSQL	Global variables
Redis	Global structures
MySQL	Global variables & global structures
Httpd	Structures
Postfix	Global variables

```
/* httpd-2.4.23/server/core.c */
...
 AP_INIT_TAKE1("ServerAdmin", set_server_string_slot,
 (void *)APR_OFFSETOF(server_rec, server_admin), RSRC_CONF,
 "The email address of the server administrator"),
...
 AP_INIT_TAKE1("ErrorLog", set_server_string_slot,
 (void *)APR_OFFSETOF(server_rec, error_fname), RSRC_CONF,
 "The filename of the error log"),
...
 static const char *set_server_string_slot(cmd_parms *cmd,
    void *dummy,
                                const char *arg)
 {
   /* This one's pretty generic... */

   int offset = (int)(long)cmd->info;
   char *struct_ptr = (char *)cmd->server;

   const char *err = ap_check_cmd_context(cmd,
                                NOT_IN_DIR_LOC_FILE);
   if (err != NULL) {
      return err;
   }

   *(const char **)(struct_ptr + offset) = arg;
   return NULL;
 }
...
```

Fig. 5. Two different configuration options in httpd.2.4.23 share the same set-function.

```
/* redis-3.2.5/src/config.c */
...
if (!strcasecmp(argv[0],"port") && argc == 2) {
      server.port = atoi(argv[1]);
      if (server.port < 0 || server.port > 65535) {
            err = "Invalid port"; goto loaderr;
      }
}
...
```

Fig. 6. Example of a configuration constraint in if-condition statement in redis.3.2.5.

to handle those situations. In our study, the majority of numeric constraints are in structures that conduct the mapping between the configuration options and the relevant configuration variables; we call them mapping constraints accordingly. In general, this kind of mapping constraint widely exists in MySQL, PostgreSQL, and Postfix. In detail, as Fig. 7 shows, uniform object declarations are used in MySQL to illustrate the constraints,

```
/* mysql-5.7.16/sql/sys_vars.cc */
...
 static Sys_var_ulong Sys_connect_timeout(
     "connect_timeout",
     "The number of seconds the mysqld server is waiting for
         a connect "
     "packet before responding with 'Bad handshake'",
     GLOBAL_VAR(connect_timeout), CMD_LINE(REQUIRED_ARG),
     VALID_RANGE(2, LONG_TIMEOUT), DEFAULT(CONNECT_TIMEOUT),
         BLOCK_SIZE(1));
...
```

(a)

```
/* postgresql-9.5.6/src/backend/utils/misc/guc.c */
...
 {
   {"geqo_effort", PGC_USERSET, QUERY_TUNING_GEQO,
    gettext_noop("GEQO: effort is used to set the default
       for other GEQO parameters."),
    NULL
   },
   &Geqo_effort,
   DEFAULT_GEQO_EFFORT, MIN_GEQO_EFFORT, MAX_GEQO_EFFORT,
   NULL, NULL, NULL
 },
...
```

(b)

```
/* postfix-3.1.3/src/global/mail_params.c */
...
 static const CONFIG_TIME_TABLE time_defaults[] = {
   VAR_EVENT_DRAIN, DEF_EVENT_DRAIN, &var_event_drain, 1, 0,
   VAR_MAX_IDLE, DEF_MAX_IDLE, &var_idle_limit, 1, 0,
   VAR_IPC_TIMEOUT, DEF_IPC_TIMEOUT, &var_ipc_timeout, 1, 0,
   VAR_IPC_IDLE, DEF_IPC_IDLE, &var_ipc_idle_limit, 1, 0,
   VAR_IPC_TTL, DEF_IPC_TTL, &var_ipc_ttl_limit, 1, 0,
   VAR_TRIGGER_TIMEOUT, DEF_TRIGGER_TIMEOUT, &
      var_trigger_timeout, 1, 0,
   VAR_FORK_DELAY, DEF_FORK_DELAY, &var_fork_delay, 1, 0,
   VAR_FLOCK_DELAY, DEF_FLOCK_DELAY, &var_flock_delay, 1, 0,
   VAR_FLOCK_STALE, DEF_FLOCK_STALE, &var_flock_stale, 1, 0,
   VAR_DAEMON_TIMEOUT, DEF_DAEMON_TIMEOUT, &
      var_daemon_timeout, 1, 0,
   VAR_IN_FLOW_DELAY, DEF_IN_FLOW_DELAY, &var_in_flow_delay,
      0, 10,
   0,
 };
...
```

(c)

Fig. 7. Examples of mapping constraints. (a) MySQL-5.7.16. (b) PostgreSQL-9.5.6. (c) Postfix-3.1.3.

especially with some macros to improve readability, while specific structure arrays are used in PostgreSQL and Postfix.

Finding 5: Various forms of enumeration constraints are used in different software packages, and are not confined simply to switch-case-statement situations.

Based on our study statistics, we found that enumeration constraints are rarely found in switch-case-statement patterns in the investigated software, as shown in Table VII. Therefore, the current works could only obtain a few enumeration constraints.

Furthermore, with further analysis, we found that the enumerative values of enumeration configuration options are always concentrated in some clustered code snippets, just as shown in

TABLE VII
NUMBERS OF ENUMERATION CONSTRAINTS THAT CAN BE EXTRACTED FROM SWITCH-CASE STATEMENTS

Software	Total	Switch-case statements
PostgreSQL	97	2
Redis	19	0
MySQL	86	3
Httpd	26	1
Postfix	19	0

```
/* postgresql-9.5.6/src/backend/utils/misc/guc.c */
...
 {
  {"backslash_quote", PGC_USERSET, COMPAT_OPTIONS_PREVIOUS,
   gettext_noop("Sets whether \"\\'\" is allowed in string
       literals."),
   NULL
  },
  &backslash_quote,
  BACKSLASH_QUOTE_SAFE_ENCODING, backslash_quote_options,
  NULL, NULL, NULL
 },
...
 static const struct config_enum_entry
    backslash_quote_options[] = {
  {"safe_encoding", BACKSLASH_QUOTE_SAFE_ENCODING, false},
  {"on", BACKSLASH_QUOTE_ON, false},
  {"off", BACKSLASH_QUOTE_OFF, false},
  {"true", BACKSLASH_QUOTE_ON, true},
  {"false", BACKSLASH_QUOTE_OFF, true},
  {"yes", BACKSLASH_QUOTE_ON, true},
  {"no", BACKSLASH_QUOTE_OFF, true},
  {"1", BACKSLASH_QUOTE_ON, true},
  {"0", BACKSLASH_QUOTE_OFF, true},
  {NULL, 0, false}
 };
...
```

(a)

```
/* httpd-2.4.23/server/core.c */
...
 static const char *set_enable_sendfile(cmd_parms *cmd,
    void *d_, const char *arg)
 {
  core_dir_config *d = d_;

  if (strcasecmp(arg, "on") == 0) {
     d->enable_sendfile = ENABLE_SENDFILE_ON;
  }
  else if (strcasecmp(arg, "off") == 0) {
     d->enable_sendfile = ENABLE_SENDFILE_OFF;
  }
  else {
     return "parameter must be 'on' or 'off'";
  }

  return NULL;
 }
...
```

(b)

Fig. 8. Examples of enumeration clustering in source code from different software packages. (a) PostgreSQL-9.5.6. (b) Httpd-2.4.23.

Fig. 8. In detail, Fig. 8(a) illustrates that a series of structural arrays are used to build the relevant relationship between the enumerative values and the enumerative string in PostgreSQL. Similar methods are used in Redis and MySQL. Fig. 8(b) illustrates that macros are used for enumeration to fulfill the assignment of the configuration value to program variable values in Apache Httpd.

Finding 6: Semantic information is implied in the parameters of functions that use configuration options.

Current research mainly uses dataflow information to trace back to a known lib-function to infer the semantic meaning of a configuration option. However, the statistics of our manual study identify that a great number of configuration options

```
/* postfix-3.1.3/src/global/mail_dict.c */
...
 dymap_init(path, var_shlib_dir);
 myfree(path);
...

/* postfix-3.1.3/src/global/dynamicmaps.c */
...
 void dymap_init(const char *conf_path, const char *
    plugin_dir)
 {
  static const char myname[] = "dymap_init";
  ...
 }
...
```

Fig. 9. Example of semantic information in the parameter identifiers from a function call in Postfix-3.1.3.

TABLE VIII
PROPORTION OF CONFIGURATION OPTIONS THAT CAN BE TRACED TO KNOWN LIB-FUNCTIONS AND FUNCTION PARAMETERS

Software	Configuration options that can be traced to known lib-functions	Configuration options that can be traced to known function parameters
PostgreSQL	28.6%	42.9%
Redis	50.0%	66.7%
MySQL	40.0%	65.0%
Httpd	33.3%	88.9%
Postfix	33.3%	44.4%

cannot be traced in this way. They are not able to be traced back to a commonly known lib-function. On the other hand, we find that the nomenclature of the parameters in the function always contains semantic information rather than arbitrary strings. Consequently, we could use the semantic information in the parameters of function calls to infer the semantic meaning of a configuration option. Taking the configuration option "shlib_directory" as an example, as shown in Fig. 9, it cannot be traced back to a known lib-function call during its usage, but it was called by function "dymap_init," where the second parameter is "plugin_dir"; thus, the semantic meaning of confiugration option "shlib_directory" can be inferred to be "DIR" based on this hint.

Based on this finding, we summarized the statistics of the aforementioned situation, as shown in Table VIII.

C. Obstacles to Extracting Configuration Constraints

Finding 7: As a consequence of the various structural forms in different kinds of software, there is no mature method for dealing with structural configuration options and the effect scope of configuration options.

In our study, we note that there are widely used structural configuration options to organize the configuration in Httpd and MySQL, but the consideration of those situations are as yet left blank. For instance, configuration keyword "<VirtualHost" is used to customize the virtual host configuration in Apache Httpd, as shown in Fig. 1. In these situations, the configuration settings only take effect in a given virtual host. MySQL also uses a similar measure, as shown in Fig. 10. Moreover, PostgreSQL uses specific parameters to limit the effect scope of configuration options, such as keyword "PGC_USERSET" in Figs. 7 and 8(a). When it comes to these parameters that have scope

```
[client]
       port = 3306
       socket = /tmp/mysql.sock
[mysqld]
       port = 3306
       socket = /tmp/mysql.sock
       basedir = /usr/local/mysql
       datadir = /data/mysql
       pid-file = /data/mysql/mysql.pid
       user = mysql
       bind-address = 0.0.0.0
```

Fig. 10. Example of the scope effect in the configuration file of MySQL-5.7.16

TABLE IX
PROPORTION OF FILE RESOURCE-RELATED CONFIGURATION THAT HAS BEEN CHECKED

Software	File resource-related configurations	Proportion of accessibility checking
PostgreSQL	14	0.0%
Redis	4	66.7%
MySQL	20	25.0%
Httpd	9	44.4%
Postfix	9	33.3%

effect in the program, it is challenging to distinguish the exact scope with rich, context-sensitive information, and complex implementation, limiting the extraction of those configuration constraints.

Finding 8: The majority of the studied software packages do not have sufficient checking mechanisms for resource-related configuration options.

Various kinds of resources are needed at runtime in a software package, such as files, network, hardware, and so on. The majority of these resources are declared in configuration files under different users' system environments. To ensure the normal functioning of the software, the accessibility of those resources should be guaranteed. In this part of our study, the *accessibility* mainly focuses on the existence of those resources. However, it is a pity that the checking of those resources is lacking generally. In terms of file resources, the proportions of accessibility checking are shown in Table IX.

In light of this situation, we speculate that software developers might think users will prepare the necessary resources and configure them properly. Nevertheless, it is a great vulnerability in configuration design and implementation.

IV. AUTOMATIC EXTRACTION OF CONFIGURATION CONSTRAINTS

Owing to the necessity of extracting configuration constraints in misconfiguration prevention and diagnosis, as well as our findings, we will propose several strategies to maximize the automatic extraction of configuration constraints from source code. To address this problem, common characteristics and patterns are summarized from various pieces of source code. To conclude, we will mainly focus on the following three kinds of configuration constraint:

1) numeric value ranges;
2) enumeration constraints; and
3) the semantic meaning of configuration options.

```
/* postgresql-9.5.6/src/include/utils/guc_tables.h */
...
 struct config_int
 {
   struct config_generic gen;
   /* constant fields, must be set correctly in initial
      value: */
   int    *variable;
   int    boot_val;
   int    min;
   int    max;
   GucIntCheckHook check_hook;
   GucIntAssignHook assign_hook;
   GucShowHook show_hook;
   /* variable fields, initialized at runtime: */
   int    reset_val;
   void   *reset_extra;
 };
...
```

Fig. 11. Definition of mapping structure in PostgreSQL-9.5.6.

A. Extracting Numeric Value Ranges From Mapping Code Snippets

As mentioned in Finding 3, the mapping structures of configuration options and relevant variables contain many value range constraints; i.e., mapping constraints. Although there are obvious features that could be used to locate the mapping code snippets [19], it is difficult to extract the constraints from those snippets without any semantic knowledge. Aiming at this challenge, we were inspired to use some semantic information to help with the extraction. For instance, the definition of the mapping structure arrays in PostgreSQL is shown in Fig. 11, in which the value range of integer type configuration is defined by structure fields "int min" and "int max." Therefore, if we could determine the meaning of these fields during the automatic analysis, the accuracy of the constraint extraction would be much improved. To achieve this goal, we implement our automatic extraction from two aspects with Clang [20], which provides a convenient API for the abstract syntax tree (AST) in C/C++.

1) Constraint Extraction Using Statistical Information: Based on the observations and methods used in ConfMapper [19], we located the main mapping source code snippets from plenty of source files. Then, we made use of the statistics for different configuration options in mapping code snippets to determine the possible value ranges of numeric options. For instance, the mapping snippets for the integer configuration options in PostgreSQL are illustrated in Fig. 12. In lines 11, 21, and 30 of Fig. 12, the "struct config_int ConfigureNamesInt" declares the "boot_val," "min," and "max" of different configuration options. It is obvious that the "min" value is always less than the "max" value, and the "boot_val" value is always between them. Hence, we used this hint to find the possible bounds of value range in the structure declaration. Specifically, when it comes to an instance of a mapping structure for configuration option, we used the following formula, which we called *relation formula*, to describe the relationship between different numeric fields: If "struct.field$_i$" and "struct.field$_j$" fulfill the formula, then we count the relationship as "1," otherwise we count it as "0."

$$\text{struct.field}_i > \text{struct.field}_j.$$

```
/* postgresql-9.5.6/src/backend/utils/misc/guc.c */
...
static struct config_int ConfigureNamesInt[] =
{
  {
    {"archive_timeout", PGC_SIGHUP, WAL_ARCHIVING,
      gettext_noop("Forces a switch to the next xlog file if
         a new file has not been started within N seconds."),
      NULL, GUC_UNIT_S
    },
    &XLogArchiveTimeout,
    0, 0, INT_MAX / 2,
    NULL, NULL, NULL
  },
  {
    {"post_auth_delay", PGC_BACKEND, DEVELOPER_OPTIONS,
      gettext_noop("Waits N seconds on connection startup
         after authentication."),
      gettext_noop("This allows attaching a debugger to the
         process."),
      GUC_NOT_IN_SAMPLE | GUC_UNIT_S
    },
    &PostAuthDelay,
    0, 0, INT_MAX / 1000000,
    NULL, NULL, NULL
  },
  {
    {"default_statistics_target", PGC_USERSET,
       QUERY_TUNING_OTHER,
      gettext_noop("Sets the default statistics target."),
      gettext_noop("This applies to table columns that have
         not had a column-specific target set via ALTER TABLE
          SET STATISTICS.")
    },
    &default_statistics_target,
    100, 1, 10000,
    NULL, NULL, NULL
  },
  ...
}
...
```

Fig. 12. Example of mapping snippets of integer configuration option in PostgreSQL-9.5.6.

TABLE X
PERCENTAGE OF DIFFERENT FIELDS THAT SATISFY THE RELATION FORMULA IN THE MAPPING STRUCTURE OF POSTGRESQL

Pctg ($field_i > field_j$)	$field_3$	$field_4$	$field_5$
$field_3$	-	37.5%	0.0%
$field_4$	0.0%	-	0.0%
$field_5$	90.9%	100.0%	-

Then, we calculated the proportion of configuration options that satisfy this formula for every pair of "struct.field_i" and "struct.field_j" in the whole mapping structure. Finally, an $n \times n$ matrix is generated, where n is the number of numeric fields in the mapping structure. The elements in the matrix, which we record as "Pctg(field$_i$ > field$_j$)," represent the proportion of configuration options that satisfy the formula; i.e., the possibility that the value in "struct.field_i" is greater than "struct.field_j." Based on this matrix, we can identify the possible relationship between different structure fields, while the upper bound is always greater than the lower bound in the value range of a configuration option.

For instance, in PostgreSQL, there are three numeric fields in the mapping structure; i.e., "boot_val," "min," and "max." We calculated the $n \times n$ matrix as shown in Table X. Based on the data in Table X, we can see that "field_5" is always greater than "field_4," which correspond to the "max" and "min" in the struct definition, respectively. Therefore, it is highly possible that "field_5" is the upper bound and "field_4" is the lower bound of the value range for configuration options in PostgreSQL.

TABLE XI
COLLECTION OF WORDS THAT REPRESENT THE UPPER AND LOWER BOUNDS

Word collection name	**Word list**
Upper bounds	max, limit, len, end, high, utmost, tail, upper, boundary, bound, restrict, confine, farthest, last, final, goal, extreme, uttermost
Lower bounds	min, start, head, begin, low, lower, first, outset, commence, kickoff

2) Constraint Extraction Using Identifier Semantic Information: There is plenty of information given in the identifier name when developers write source code, helping understand, improve, and maintain the software. As Liu *et al.* [21] mentioned, if we do not make use of the meaning of identifier, when running an analysis on a human-written program with meaningful identifier names and on an equivalent program where all identifiers are consistently replaced with arbitrary strings, it gives exactly the same result. We observe that the identifiers of the value ranges in the mapping structure always use meaningful items, such as "min" and "max," "min_val" and "max_val." Based on these observations, we made use of items or words that represent the boundaries to identify the possible value ranges of numeric configuration options.

To achieve this, we summarized the frequently used words and items that represent the upper and lower bounds. To remedy the insufficiency of human experience, we used WordNet [22] to expand our collection of words and items, which is a lexical database established by Princeton University that groups English words into sets of synonyms, and records relations among these synonym sets or their members. The final collection of words and items representing the upper and lower bounds is listed in Table XI.

Based on the collection of words, we checked every field identifier in the definition of the mapping structure; if the identifier contains any of the words, we mark the weight of this identifier as "1," otherwise as "0." As some of the identifiers might be a joining of several words, so we handled this situation by splitting the identifier using the method in ConfMapper [19], and checked the split word against the word collection. Finally, for every identifier, we determined a weight that represents its meaning, called *IdenWeight*, where *IdenWeight*$\in \{0, 1\}$.

3) Configuration Constraints Recommendation: Based on the two previous steps, we have gotten different aspects of information for the value ranges in the mapping structure. Next, we combined these two aspects to be able to recommend the possible fields that represent value ranges in the mapping structure. To address this problem, we defined the confidence_level to represent the possibility of a field being the upper/lower bound of a value range. The *confidence*_level is illustrated as follows:

$$\begin{cases} \text{confidence}_i = \alpha \cdot \text{stat\_ratio}_i \\ \qquad\qquad + (1-\alpha) \cdot \text{iden\_ratio}_i \\ \text{stat\_ratio}_i = \frac{1}{n} \cdot \sum_{j=0, j\neq i}^{n} \text{Pctg}(\text{field}_i > \text{field}_j) \\ \text{iden\_ratio}_i = \text{IdenWeight}(\text{field}_i) \end{cases} \quad (1)$$

where i and j are the sequence numbers of fields in mapping structure, confidence$_i$ is the confidence_level value of the ith

TABLE XII
CONFIDENCE LEVEL FOR DIFFERENT FIELDS IN THE MAPPING STRUCTURE OF POSTGRESQL

Confidence level	Upper bound	Lower bound
$field_3$	0.093775	0.099
$field_4$	0	**0.97725**
$field_5$	**0.97725**	0

numeric field, n in stat_ratio$_i$ is the total number of numeric fields in the mapping structure, α is an adjustable parameter for controlling the effect of the two aspects mentioned in the previous two steps. After conducting experiments with different α values, we recognize the α value of 0.5 as providing a better effect.

Based on the definition of confidence_level, we chose the most likely upper bound and lower bound for a numeric configuration option. For instance, the confidence_level of the mapping structure in PostgreSQL is shown in Table XII. It is obvious that field$_5$ is the most likely upper bound of the configuration option, and field$_4$ is the most likely lower bound.

B. Extracting Enumeration Constraints From Clustered Code Snippets

For the enumeration constraints, we used the clustered code snippets to extract the enumeration constraints mentioned in Finding 5. If we locate the clustered code snippets, as Fig. 8 shows, the uniformed structure and characteristics provide much help in extracting enumeration constraints. Hence, our method includes two steps for extracting enumeration constraints: 1) locate the clustered code snippets in the source code; and 2) extract the enumeration constraints from the clustered code snippets.

1) Locating the Clustered Code Snippets for the Enumeration Options: For enumeration configuration options, it is common that they are usually used in assignments or conditional statements with their possible enumerative value used in a discrete way. Hence, if we find the usage of the program variables for enumeration options to get their possible values, then observe the distribution of these enumerative values, the clustered code snippets can be located.

On the basis of the methods used in ConfMapper [19], we were able to determine the program variables of the relevant configuration options. The next step was to find the usage of the enumeration option in the source code. For a simple assignment statement, it is easy to obtain the enumerative value. For conditional statements, such as the conditions in if statements and switch case statements, they always consist of several complex conditions. Taking Fig. 13 as example, the usage of the enumeration option variable "server.maxmemory_policy" is in a complex context. To address this situation, we used a binary condition tree (BCT) to represent the statement and obtain the enumerative value we want. For the statement shown in Fig. 13, the corresponding BCT is shown in Fig. 14. Then, based on BCT, we were able to extract the usage of the enumerative value from the leaf node, shown as the blue node in Fig. 14. The switch-case situation could be handled in a similar way.

```
/* redis-3.2.5/src/object.c */
...
if ((server.maxmemory == 0 ||
    (server.maxmemory_policy !=
       REDIS_MAXMEMORY_VOLATILE_LRU &&
     server.maxmemory_policy !=
       REDIS_MAXMEMORY_ALLKEYS_LRU)) &&
    value >= 0 && value < REDIS_SHARED_INTEGERS){
    ...
}
...
```

Fig. 13. Example of a conditional statement in Redis-3.2.5.

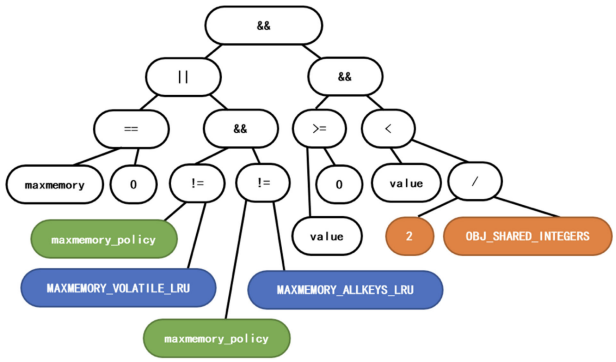

Fig. 14. BCT of the example conditional statement in Redis-3.2.5.

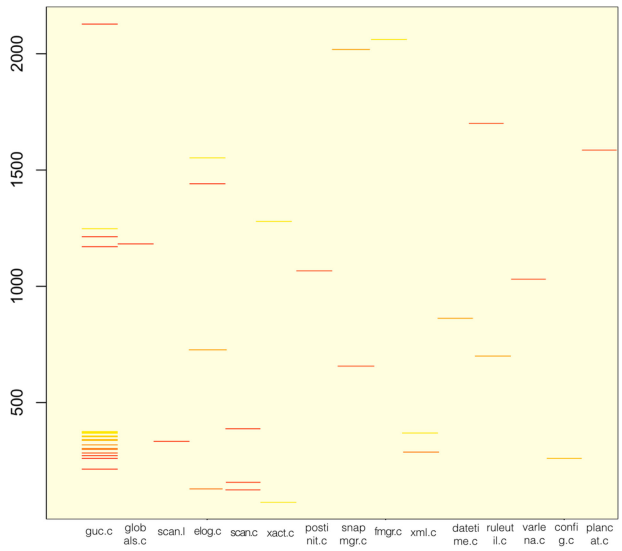

Fig. 15. Distribution of the appearance of enumerative values in PostgreSQL-9.5.6.

After we had identified the enumerative values of the enumeration configuration options, it was easy to find the location distribution of all enumerative values used and confirm the clustered code snippets. For instance, the distribution in PostgreSQL is illustrated in Fig. 15. The horizontal axis is the source file in the PostgreSQL projects, and the vertical axis is the line number where the enumerative value appears. From Fig. 15, we can see that these enumerative values are mainly clustered in one single file "guc.c," so we analyzed this "guc.c" file to get the enumeration constraints.

```
/* postgresql-9.5.6/src/config.c */
...
static const struct config_enum_entry
    log_error_verbosity_options[] = {
        {"terse", PGERROR_TERSE, false},
        {"default", PGERROR_DEFAULT, false},
        {"verbose", PGERROR_VERBOSE, false},
        {NULL, 0, false}
};

static const struct config_enum_entry log_statement_options
    [] = {
        {"none", LOGSTMT_NONE, false},
        {"ddl", LOGSTMT_DDL, false},
        {"mod", LOGSTMT_MOD, false},
        {"all", LOGSTMT_ALL, false},
        {NULL, 0, false}
};
...
```

Fig. 16. Example of the mismatch of enumeration value spaces in PostgreSQL-9.5.6.

2) Analyzing the Clustered Code Snippets With Textual Analysis and Structural Information: Based on the previous step, we then used textual analysis to find the possible enumerative strings, i.e., the value spaces of the enumeration configurations. In the clustered code snippets, the enumerative string always comes with an enumerative value, and is surrounded by double quotation marks. Therefore, we were able to determine get the enumerative string and value pair by using textual matching and a similarity comparison. In detail, we found the double quotation marks first, then calculated the similarity between this string in the double quotation marks and the enumerative value in several neighboring lines, and, finally we chose the enumerative value with the highest similarity to form an enumerative pair "<string, value>."

However, there might be some mismatches for these enumerative pairs. Without syntactic or semantic information, the textual analysis might mismatch the enumerative pair of another configuration option to this configuration option. For instance, as shown in Fig. 16, the enumeration constraints of configuration option "log_statement" defined in struct "log_statement_options" are "{none, ddl, mod, all}," but we might add "verbose" or even "default" in struct "log_error_verbosity_options" into the enumeration constraints of "log_statement" if we choose a large analysis scope in the textual analysis (if we choose a small analysis scope, some enumerativ pair might not be able to be covered). As a consequence, we used AST to verify the correctness of those enumerative pairs. In detail, if the majority of those enumerative pairs shared the same common parent of AST type, we treated this as their *subancestor AST*, and recorded the position of the enumerative value and string in the *subancestor AST*. For those pairs that do not share same parent AST type, we used the common position of the enumerative string to find its *subancestor AST* and determined the corresponding enumerative value, respectively. For example in PostgreSQL, one enumerative pair of configuration option "xmloption" is "<content, XMLOPTION_CONTENT>," as shown in frames in Fig. 17. If we found that the majority of those enumerative pairs shared the same *subancestor AST* as "InitListExpr" and were in a corresponding position, then we were able to find other mismatched enumerative pairs with this information, and identify the corresponding enumeration

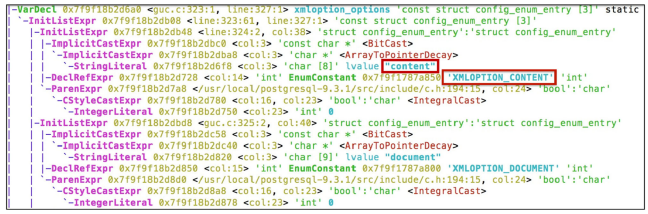

Fig. 17. Example of AST structure of enumeration clustered code snippets in PostgreSQL-9.5.6.

constraints. For those mismatch situations, because they are in different sub-ASTs, so we could exclude them from the correct enumeration constraints. In this example, we determined the enumeration constraint of configuration option "xmloption" to be "{content, document}."

C. Extracting Semantic Meaning Using Function Parameter Information

To extract the semantic meaning of the configuration options, we enhanced the dataflow analysis mentioned in SPEX [14] by adding some semantic information. The detailed procedures are illustrated in the following section.

1) Tracing the Configuration Variable Usage in the Source Code: First, we established a dictionary based on our experience, consisting of some commonly used parameters or abbreviations and their associated semantic meaning. For example, if a parameter in a function call is "filepath," we think that it has a semantic meaning of "PATH." Similarly, another parameter "root_dir" may have "DIR" as its semantic meaning.

Then, based on this prebuilt dictionary, we conducted a dataflow analysis, which started from the usage of the configuration variables in the function call. Once a configuration variable had been used in a function, we traced into the function implementation and replaced the traced configuration variable with the parameter of this function in the corresponding position. This trace was ended when we traced back to an already known lib-function or a parameter in the prebuilt dictionary mentioned previously. Based on this procedure, we could determine the semantic meaning of a configuration option. For instance, in Fig. 9, variable "var_shlib_dir" contains the value of configuration option "shlib_directory" in Postfix. When we traced the use of the variable "var_shlib_dir," we first traced the function call "dymap_init" to find its implementation. Then, in function "dymap_init," the configuration variable was used as its second argument, so we replaced variable "var_shlib_dir" with parameter "plugin_dir." Then, we looked in our prebuilt dictionary to find parameter "plugin_dir," and found that "plugin_dir" has a "DIR" semantic meaning. Hence, we concluded that the semantic meaning of configuration option "shlib_directory" is "DIR."

D. Experiments and Evaluation

In this section, we will conduct experiments to evaluate the effectiveness of our methods in extracting configuration constraints. In detail, there are three experiments with respect to the

TABLE XIII
NUMERIC VALUE RANGES EXTRACTED FROM MAPPING CODE SNIPPETS

Software	Numbers of value ranges extracted	Proportion that were successfully extracted
PostgreSQL	100	100.0%
MySQL	170	85.9%
Postfix	31	100%
MariaDB	210	86.2%
Percona-Server	195	90.8%

TABLE XIV
ENUMERATION CONSTRAINTS EXTRACTED FROM CLUSTERED CODE SNIPPETS

Software	Numbers of enumeration constraints extracted	Proportion that were successfully extracted
PostgreSQL	22	89.5%
MySQL	24	100.0%
Redis	4	100.0%
Httpd	20	80.0%
Nginx	48	87.5%
Percona-Server	25	100.0%
MariaDB	20	100.0%
Git	8	62.5%

TABLE XV
SEMANTIC MEANING EXTRACTED WITH STRATEGY 3

Software	Numbers of semantic type newly added	Proportion that were successfully extracted
PostgreSQL	1	100.0%
MySQL	10	90.0%
Redis	2	100.0%
Httpd	5	80.0%
Postfix	1	100.0%

aforementioned three strategies. Based on the code features we mentioned in the findings of Section III, we select the software projects in Tables XIII, XIV, and XV to conduct our experiments. To evaluate the correctness of the extracted results, we manually analyzed the related code snippets in the target software projects, and extracted the corresponding constraints as the constraints oracle. Then, we checked the results of automated tools manually to calculate the proportion of constraints that were successfully extracted.

1) Experiment on Extracting Numeric Value Ranges From Mapping Code Snippets: We did an experiment on other software packages with regard to extracting numeric value ranges from mapping code snippets, and the results illustrated in Table XIII show the effectiveness of our method in extracting the mapping constraints.

2) Experiment on Extracting enumeration Constraints From Clustered Code Snippets: Based on the method mentioned about extracting enumeration constraints from clustered code snippets, we performed experiments on eight software packages. In addition to the five software packages in the main study, we also used Percona-Server, MariaDB, and Git in our experiment. The results in Table XIV show the effectiveness of our automation.

3) Experiment on Extracting Semantic Meaning Using Function Parameter Information: Based on the aforementioned procedure concerning extracting semantic meaning of a configuration option, we conducted an experiment to evaluate the effectiveness of our method. The final results are listed in Table XV. From the statistics, we can see that this method can increase the success rate in semantic-meaning determination in most software packages.

V. RELATED WORK

This section introduce the related work about misconfiguration diagnosis.

Nowadays, misconfigurations has become one of the major causes of software failures. Faced with such situations, many researchers have worked hard to tackle these problems. In general, the research mainly contains the following directions.

Configuration constraint-related research: Some work has been done regarding configuration constraints in software. ECC Fixer [10] uses Reiter's theory of diagnosis to transfer the problem of configuration options' value range to a satisfiability problem (i.e. SAT problem). Based on eCOS, i.e., embedded configurable operating system, ECC Fixer does not need to extract configuration constraints. SPEX [14] uses static program analysis to extract configuration constraints from source code, mainly targeting if-statement checking situations. Then, based on these configuration constraints, a tool named SPEX-INJ is proposed to inject misconfigurations into software, and expose vulnerabilities in configuration handling and software design.

Misconfiguration judgement: Nowadays, it is hard to judge whether a software failure is caused by misconfiguration or software bugs, and few studies have focused on this field. Based on massive system bug reports, [23] used feature selection techniques, such as information gain and Chi-square, to select significant terms. Then, a classifier toward bug reports was established based on different kinds of text mining algorithms. Finally, when a new bug is reported, using the classifier could quickly judge whether it is a misconfiguration report or not. Furthermore, some researchers have tried to generate misconfiguration to evaluate the response of the software system, thus, enhancing the ability of misconfiguration judgement. ConfErr [24] uses a human error model to simulate human mistakes in configuration, e.g., typo, copy-paste mistake, and other generic alternations. However, ConfErr is not guided by the configuration constraints, and it can only generate deficient misconfigurations, which impedes the analyses on system reactions. ConfDiagDetector [25] injects misconfigurations into the software under test, monitors the software outcomes under the injected misconfigurations, and uses natural language processing to analyze the output diagnostic message caused by each misconfiguration. Based on the analysis results, software developers could improve the error handling of misconfigurations for better diagnosis.

Misconfiguration Localization: When misconfiguration happens, it is vital to find out the specific configuration option causing the error; we call this misconfiguration localization. Aiming at Windows register errors, Snitch [9] monitors the read/write events in registers to build a binary decision tree between the software system and the registers. Then, based on this decision tree, users can quickly locate the error register

items. ConfDiagnoser [26] combines static analysis, dynamic analysis and compare-based methods to diagnose misconfigurations. First, ConfDiagnoser uses thin slicing to collect the set of program statements that are affected by configuration options. Then, based on instrumentation techniques, ConfDiagnoser traces the runtime route to represent the current system characteristics. Finally, through comparing with the known misconfiguration's system character, ConfDiagnoser can locate the suspicious configuration options. ConfDebugger [8] also uses forward and backward static program analysis to diagnose misconfigurations. Specifically, ConfDebugger collects the set of program statements that are affected by configuration options, which is called forward analysis, and collects the set of program statements that could be traced back from stack information, which is called backward analysis. Finally, the configuration options that affect the statements in the intersection of these two sets are the probable error configuration options. Considering latent misconfiguration, Xu *et al.* [7] extracted the executable code snippets about configuration options from source code. Then, by prerunning these code snippets and evaluating the system response, they could check the latent misconfigurations in the initialization phase of the software.

VI. Conclusion

Misconfigurations have increasingly become one of the common causes of software failure. To diagnose misconfigurations, configuration constraints have a vital influence. On the basis that it is urgently necessary to extract configuration constraints, a comprehensive study was carried out concerning the existence and variance of configuration constraints in source code. Based on the study's results, three aspects of findings were summarized in respect to the configuration constraints and configuration designs, namely the general statistics, the general features of specific kinds of constraints, and the obstacles to the automatic extraction of configuration constraints. Finally, we proposed three strategies to improve the automatic extraction of three types of configuration constraints from source code, i.e., extracting numeric value ranges from mapping code snippets, extracting enumeration constraints from clustered code snippets, and extracting semantic meaning using function parameter information. The experiments' results show the effectiveness of our strategies.

References

[1] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proc. ACM Symp. Oper. Syst. Principles*, 2011, pp. 159–172.

[2] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures Comput. Archit.*, vol. 8, no. 3, pp. 1–154, 2009.

[3] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE Softw.*, vol. 30, no. 4, pp. 88–94, Jul./Aug. 2013.

[4] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in internet services," in *Proc. Conf. Symp. Oper. Syst. Des. Implementation*, 2004, pp. 61–76.

[5] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *Proc. Conf. Usenix Symp. Internet Technol. Syst.*, 2003, pp. 165–171.

[6] A. Team, "Summary of the amazon EC2 and amazon RDS service disruption in the us east region. amazon web services," 2011. [Online]. Available: http://aws.amazon.com/message/65648

[7] T. Xu *et al.*, "Early detection of configuration errors to reduce failure damage," in *Proc. USENIX Symp. Oper. Syst. Implementation*, 2016, pp. 619–634.

[8] Z. Dong, M. Ghanavati, and A. Andrzejak, "Automated diagnosis of software misconfigurations based on static analysis," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops*, 2013, pp. 162–168.

[9] J. Mickens, M. Szummer, and D. Narayanan, "Snitch: Interactive decision trees for troubleshooting misconfigurations," in *Proc. Usenix Workshop Tackling Comput. Syst. Problems Mach. Learn. Tech.*, 2007, pp. 1–8.

[10] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 58–68.

[11] MySQL, 2017. [Online]. Available: http://www.mysql.com/

[12] Httpd, 2017. [Online]. Available: http://httpd.apache.org/

[13] T. Xu, L. Jin, X. Fan, R. Talwadker, R. Talwadker, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proc. Joint Meeting Found. Softw. Eng.*, 2015, pp. 307–319.

[14] T. Xu *et al.*, "Do not blame users for misconfigurations," in *Proc. ACM Symp. Oper. Syst. Principles*, 2013, pp. 244–259.

[15] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *Proc. Int. Conf. Softw. Eng.*, 2011, pp. 131–140.

[16] J. Zhang *et al.*, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2014, pp. 687–700.

[17] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Comput. Surveys*, vol. 47, no. 4, pp. 1–41, 2015.

[18] W. Li, S. Li, X. Liao, X. Xu, S. Zhou, and Z. Jia, "Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation," in *Proc. Int. Conf. Eval. Assessment Softw. Eng.*, 2017, pp. 88–97.

[19] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, "Confmapper: Automated variable finding for configuration items in source code," in *Proc. Int. Conf. Softw. Quality, Rel. Security Companion*, 2016, pp. 228–235.

[20] Clang, 2017. [Online]. Available: http://clang.llvm.org/

[21] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 1063–1073.

[22] WordNet, 2017. [online]. Available: http://wordnet.princeton.edu/

[23] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou, "Automated configuration bug report prediction using text mining," in *Proc. IEEE Comput. Softw. Appl. Conf.*, 2014, pp. 107–116.

[24] L. Keller, P. Upadhyaya, and G. Candea, "Conferr: A tool for assessing resilience to human configuration errors," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS DCC*, 2008, pp. 157–166.

[25] S. Zhang and M. D. Ernst, "Proactive detection of inadequate diagnostic messages for software configuration errors," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 12–23.

[26] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 312–321.

[27] S. Zhou *et al.*, "Easier said than done: Diagnosing misconfiguration via configuration constraints analysis," in *Proc. Int. Conf. Eval. Assessment Softw. Eng.*, 2017, pp. 196–201.

Xiangke Liao (M'15) received the B.S. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 1985, and the M.S. degree from the National University of Defense Technology, Changsha, China, in 1988.

He is currently a Full Professor and the Dean of the School of Computer Science, National University of Defense Technology. His research interests include parallel and distributed computing, high-performance computer systems, operating systems, software reliability, cloud computing, and networked embedded systems.

Prof. Liao is a member of the ACM.

Shulin Zhou received the B.S. and M.S. degrees from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2014 and 2016, respectively. He is currently working toward the Ph.D. degree at the National University of Defense Technology.

His main research interests include software engineering, software reliability, operating system, and so on.

Mr. Zhao is a student member of the ACM.

Shanshan Li received the M.S. and Ph.D. degrees from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2003 and 2007, respectively.

She was a visiting scholar at Hong Kong University of Science and Technology in 2007. She is currently an Associate Professor with the School of Computer Science, National University of Defense Technology. Her main research interests include software engineering, software reliability, and operating system.

Dr. Li is a member of the ACM.

Zhouyang Jia received the B.S. and M.S. degrees from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2013 and 2015, respectively. He is currently working toward the Ph.D. degree at the National University of Defense Technology.

His main research interests include software engineering, software reliability, operating system, and so on.

Mr. Jia is a student member of the ACM.

Xiaodong Liu received the M.S. and Ph.D. degrees from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2009 and 2014, respectively.

He is currently an Assistant Professor with the School of Computer Science, National University of Defense Technology. His main research interests include software engineering, software reliability, operating system, and so on.

Dr. Liu is a student member of the ACM.

Haochen He received the B.S. degrees from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2017. He is currently working toward the Ph.D. degree at the National University of Defense Technology.

His main research focuses on software engineering.