# ConfMapper: Automated Variable Finding for Configuration Items in Source Code

Shulin Zhou [†], Xiaodong Liu [†], Shanshan Li [†], Wei Dong [†] Xiangke Liao [†], Yun Xiong [†]
[†] National University of Defense Technology, Changsha, P. R. China
Fudan University, Shanghai, P. R. China
{zhoushulin, liuxiaodong, shanshanli, wdong, xkliao }@nudt.edu.cn
yunx@fudan.edu.cn

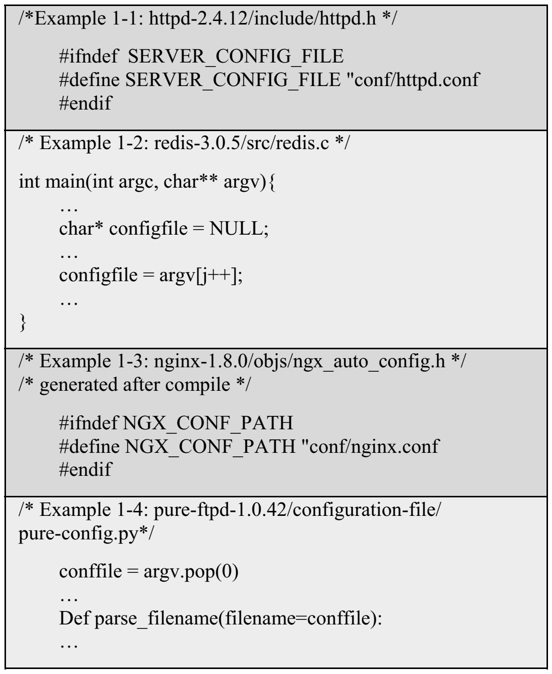
***Abstract*—Misconfigurations have become the major cause of software failures as a consequence of software development. Traditional methodology of misconfiguration diagnosis based on program analysis has distinct advantages in accuracy and efficiency. However, the state-of-the-art researches do the mapping between the configuration items and relevant program variables either manually locating or following specified annotation with fixed format, which requires enormous domain knowledge and heavy workload. In order to address the challenges, we manually studied eight popular open-source software and observed some mapping-related features. Based on these observations, we proposed a tool named ConfMapper to accomplish the automated mapping from the configuration items to the relevant program variables without understanding the complicated semantic context in source code. We carried out comprehensive experiments upon seven popular open-source software. ConfMapper could reach nearly 100% accuracy in mapping the configuration items to program variables, and mine about 91.5% potential configuration items in average.**

***Keywords—misconfiguration diagnosis; automated analysis; configuration items mapping***

## I. Introduction

In recent years, misconfigurations have become the major cause of software failures as a consequence of software development. As Barroso and Hoelzle[1] mentioned, misconfiguration were the second major cause of service outage in Google's main services, counting nearly 28 percentage. What's more, Rabkin and Katz[2] indicated that, considering both the reported clients' failure cases number and the total technique support time, misconfiguration was the leading cause of hadoop cluster failures.

The state-of-the-art methodology of misconfiguration diagnosis are various, among which the program analysis based diagnosing methodology has distinct advantages in accuracy and efficiency. One essential yet challenging step in program analysis based misconfiguration diagnosis is mapping between the configuration items defined in the configure file and program variables hidden in source code. Unfortunately, in the current research, the process of mapping is done either manually[3][4][5] or following specified annotation with fixed format[6], which requires enormous domain knowledge and heavy workload. Therefore, if we could automate the procedure of mapping configurations option to relevant program variables, the efficiency of misconfiguration diagnosis will be significantly improved.



```
/*Example 1-1: httpd-2.4.12/include/httpd.h */

    #ifndef SERVER_CONFIG_FILE
    #define SERVER_CONFIG_FILE "conf/httpd.conf
    #endif
```

```
/* Example 1-2: redis-3.0.5/src/redis.c */

int main(int argc, char** argv){
    …
    char* configfile = NULL;
    …
    configfile = argv[j++];
    …
}
```

```
/* Example 1-3: nginx-1.8.0/objs/ngx_auto_config.h */
/* generated after compile */

    #ifndef NGX_CONF_PATH
    #define NGX_CONF_PATH "conf/nginx.conf
    #endif
```

```
/* Example 1-4: pure-ftpd-1.0.42/configuration-file/
pure-config.py*/

    conffile = argv.pop(0)
    …
    Def parse_filename(filename=conffile):
    …
```

Fig 1. Examples of definition of the configure files in popular open-source software. Example 1-1 defines configure file path as a macro; example 1-2 declares the configure file path as a local variable; example 1-3 generates a new header file to define the specific file path as configure file path; example 1-4 handles configure file in script.

Intuitively, to build the mapping, one may want to find the program point of reading configure files, and then tracing the information flow to locate the assignment statement from specific configuration item value to relevant program variables. While in fact, this is often not the case. We manually analyzed eight popular open-source software, namely Apache Httpd, MySQL, Redis, Nginx, Postfix, Lighttd, Pure-ftpd and PostgreSQL, and find that complicated semantic context and various code style are serious obstacle in the mapping construction process. As shown in Fig. 1, in terms of the definition and declaration of configure files, various strategies have been applied in current open source software: 1) Define configure file path as a macro; 2) Declare configure file path as a local variable and wait for user's input; 3)

Generate a new header file to define the specific file path as configure file path; 4) Handle configure file in script. Besides, in the information flow from reading configure file to assigning configure value to program variables, a plenty of function pointers have been used, greatly increasing the difficulty to analyze the relevant variable locations. Therefore, more efficient and lightweight methodology should be proposed to solve the challenges mentioned above.

In order to address the above challenges, some approaches are designed based on the observation as follows. As we observed in eight popular open source software, the mapping practice from configuration items in configure file to relevant program variables is clustered in specific source files and snippets. What's more, the mapping practice in one software usually uses the formatted program structure and the naming mode between configuration items, and the relevant program variables are similar, making it possible to mapping configuration items to relevant program variables by learning the specific patterns in those files and code snippets.

Therefore, we propose a tool named ConfMapper to accomplish the automatic mapping from the configuration items to relevant program variables, without understanding the complicated semantic context in source code. In summarize, twofold contributions are made in this paper. In the first place, with manual analysis on eight popular open source software, we summarize the common patterns of mapping the configuration items to relevant program variables, inspiring the solution to do the mapping automatically. What's more, with features extracted ahead, we design and implement ConfMapper, an automation tool, to learn the mapping patterns in different open-source software, and automatically map the configuration items to relevant program variables, as well as mining the potential configuration items that are not preset in configure files. In practice, ConfMapper could reach nearly 100% accuracy in mapping the configuration items to program variables, and mine about 91.5% potential configuration items in average.

The rest of this paper is organized as follows. Section II introduces the related work. Section III briefly describes the architecture of ConfMapper, while Section IV presents the detailed design and implementation. We evaluate the design and results by extensive experiments, and present the results in Section V. Finally, we conclude this work in Section VI.

## II. Related Work

The state-of-the-art methodology of misconfiguration diagnosis can be classified into five categories, namely program analysis based methods, statistics based methods, replay based methods, comparison based methods, and hybrid methods.

The program analysis based methods mainly focus on the statements about configuration items in the software. Through dataflow analysis and control flow analysis, they can get the statements influenced by configuration values, and the exact execution path of the software. If there were any misconfiguration, the users or developers could locate the statements that went wrong rapidly and accurately. The main researches use program analysis contains Conf_Analyzer[3], ConfDebugger[4], SPEX[6], ConfAid[5], X-ray[7] and so on. However, these researched need either manual analysis or specific annotation to find the mapping relationship between configuration items and relevant program variables, requiring enormous domain knowledge and heavy workload.

The statistics based methods collect the historical information about the system behaviors, status and events as initial data. Then they mine the access patterns of configuration items in the target software based on probability statistics and machine learning. On the basis of these access patterns, a series of rules are set in accessing the configuration items. When the software system is outage, the rules that the system breaks indicate the root causes. Considering the features of statistics, mass historical data is needed to mine the accurate access patterns and rules. CODE[ 8 ], EnCore[ 9 ] and Snitch[10] are the main researches based on statistics.

The replay based methods rely on the replay techniques to observe the status changes of software system in a sandbox when the configuration items are changed. As a consequence of the features of sandbox, the multiple changes of the configuration items will not affect the system status. AutoBash[11], Triage[12] use the replay based methods to diagnose the misconfigurations.

The comparison based methods use a series of samples and criterions to diagnose misconfiguration. These samples and criterions commonly record the status or signatures of the system. The signatures and status could be from the normal system, with which the diagnosis is based on the differential of the normal system in the samples and the erroneous system. In addition, the signatures and status could be from some systems with known misconfigurations, with which the diagnosis could be done by finding the same sample while comparing with the samples and criterions. This kind of methods need abundant historical data to build the series of samples and criterions. The representative methods are Strider[13][14], PeerPressure[15][16] and Chrnous[17].

The hybrid methods are commonly combinations of the techniques mentioned above, avoiding the weakness of single method and techniques. The hybrid methods for misconfiguration diagnosis contains ConfDiagnoser[ 18 ][ 19 ], ConfSuggester[20] and so on.

## III. Architecture of ConfMapper

This section presents the main architecture of ConfMapper. An overview of ConfMapper is illustrated in Fig. 2. The whole tool consists of three phase: extraction of key-value information about configuration items, pattern learning of mapping module in source code, and verification of the chosen variables.

In the first phase, we take configure files as input, which could be original template configure file or user-specified ones. After parsing configure file on the basis of augeas[23] library API, we get the configuration items names and relevant values. Then, the above extracted information is used as a reference to find the mapping practice and learn the mappi-

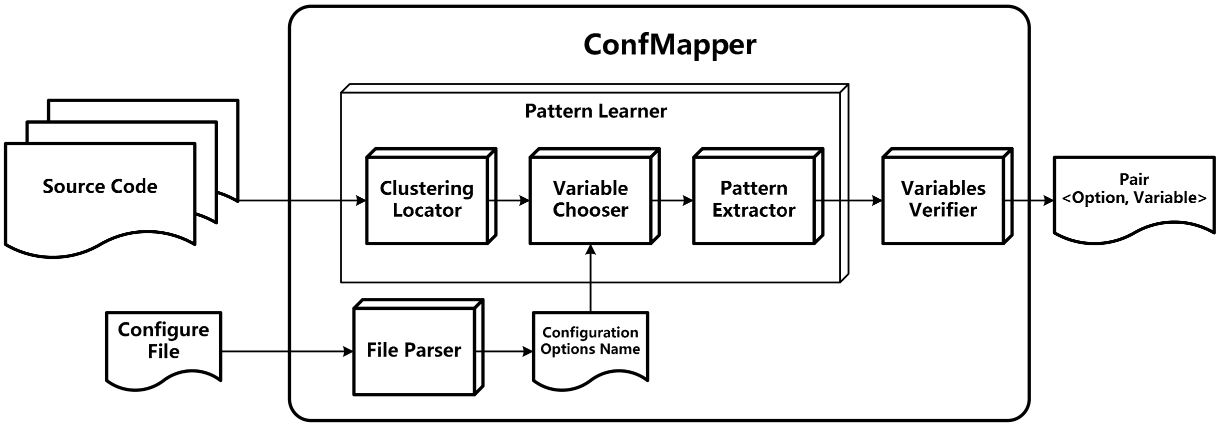


Fig 2. Architecture of ConfMapper

ng patterns in source code. This procedure includes the following three jobs: 1) locate the configuration items clustered positions in source files and snippets; 2) analyze the code snippets and choose relevant program variables; 3) extract common mapping patterns based on the chosen variables. Last but not least, with the AST of the source code, we could easily verify whether these chosen variables are real program in variables source code or just some function calls. For those function call to set the value of target variables, we could locate the relevant variable based on dataflow analysis.

We implement the analysis work on the basis of statistical analysis, and Clang[21], a C language family frontend of LLVM[22], for abstract syntax tree analysis, mapping pattern extraction and dataflow analysis.

## IV. Design and Implementation

Detailed descriptions of the design and implementation of ConfMapper is presented in this section. As Fig. 2 shows, ConfMapper consists of four main components, representing the four step to map configuration items to program variables. At first, some observations about configuration mapping practice in real-world open-source code are presented.

### A. Observations From Real-World Software

On the purpose of mapping configuration items to relevant program variables automatically, we manually studied the mapping practice in eight popular open-source software such as MySQL, Redis and PostgreSQL, and summarized several characteristic as follows.

First, the configuration item names in configure files are usually made up of single readable word or common abbreviation connected by some separators. For instance, in MySQL configuration item names use underline to connect several words, e.g. “key_buffer_size”, while in Apache Httpd, all configuration item names use camel case, i.e. uppercasing the first letter of each word, to distinguish every single word, e.g. configuration item name “DocumentRoot”. These kinds of naming mode make it easier for us to match the relevant program variables.

Then, the mapping from configuration items to program variables are commonly clustered in some specific snippets

```
/* Example 3-1：PostgreSQL-9.3.5/src/backend/utils/misc/guc.c */
…
static struct config_bool ConfigureNameBool[] =
…
static struct config_int ConfigureNameInt[] =
{
        …
        {
                {"post_auth_delay", …},
                &PostAuthDelay, …
        },
        …
}
…
static struct config_real ConfigureNameReal[] = …
…
static struct config_string ConfigureNameString[] = …
…
static struct config_enum ConfigureNameEnum[] = …
…
```

```
/* Example 3-2：MySQL-5.6.17/sql/sys_vars.cc */
…
static  Sys_var_ulong  Sys_ft_max_word_len(
        "ft_max_word_len", …
        READ_ONLY GLOBAL_VAR(ft_max_word_len), …
}
…
```

```
/* Example 3-3：redis-3.0.5/src/redis.c */
…
void loadServerConfigFromString(char* config){
        …
        if(!strcasecmp( argv[0], “tcp-keepalive”) && argc ==2){
                server.tcpkeepalive = atoi(argv[1]);
                …
        }
        else if(…)
        …
}
…
```

Fig 3. Examples of mapping practices. Example 3-1 uses five similar structures to do the mapping; example 3-2 uses serials of sys_* function pointers; example 3-3 uses if/else statements.

in one or several source files. As Fig. 3 shows, PostgreSQL does the mapping in file “guc.c”, MySQL in file “sys_vars.cc”, while Redis does it in file “config.c”. What’s more, the occurrences of configuration item names in other files are few. That observation makes sense, because the configuration item names are objects outside the source code. So only in the mapping module, the configuration items could be clustered.

Besides, it is obvious that the mapping practices in different software are commonly deployed in software-specific framework. In Fig. 3, PostgreSQL uses five similar types of Structure, namely config_int, config_real, config_bool, config_string and config_enum, to mapping different configuration items with different types to program variables. MySQL use serials of sys_* function pointers to set the program variables to the relevant configuration items' value. And Redis uses standard if/else statements to assign configure values of different configuration items to program variables.

At last, as a consequence of requirements for high readability and maintainability, programmers habitually name the program variables in a highly similar way with the configuration items. The common name approaches are: 1) keep consistency with relevant configuration item names, e.g. program variable "ft_max_word_len" and configuration item "ft_max_word_len" in MySQL; 2) eliminate the connector between single word, such as underline, and then uppercase the first letter of each word, or vice versa, e.g. program variable "DeadlockTimeout" and configuration "deadlock_timeout" in PostgreSQL; 3) add, remove or replace some letters or words, e.g. program variable "tcp-keepalive" and relevant configuration item "server.tcpkeepalive" in Redis; 4) hybrid approach among all above.

Based on the observations above, we could do the mapping from configuration items name to program variables in a lightweight approach by statistical analysis and text processing. Next we will give a detailed explanation of our approach in three steps.

## B. Parse Configure Files

With the purpose of mapping configuration items to program variables, we should know what configuration items the software has at first. Nowadays configure file is commonly used in Linux open source software for users to adjust and manage their own software system in a perfect status. Usually in C/C++ software, the configure file is written in plain text coded by ASCII. Therefore, it is necessary to analyze the plain text to extract determined configuration items names. In terms of extraction towards various software, Simple string manipulation may not work well considering the structure among different configuration items. An efficient and pervasive configure file parser is needed.

Augeas[23] is a configuration editing tool as well as a configuration API provided by a C library. It provides parsing templates for most popularly open-source software, supporting to transform the plain text configure file into canonical tree representation. With the configure tree representation we could get configuration item names easily without concentrating on the chaotic configure file format.

## C. Learning Mapping Patterns

The main part of automatically mapping the configuration items to the relevant program variables is Learning Mapping Patterns. Considering the complicated semantic context and various code style in different software, there is no common mechanism for us to do the regular program analysis. Therefore, we should learn the mapping patterns in higher level with less semantic information.

Take the observations mentioned above into consideration, it is sensible to locate the specific files and snippets in those files that do the mapping practice firstly, and then analyze the snippets to choose the most probable variables based on the similarity between configuration items and relevant program variables. Therefore, we divide our mapping pattern learning into three phases.

*1) Locate the file with mapping practice:* As we studied, there are specific files in charge of mapping the configuration items to the relevant program variables. So it makes sense that we take the occurrences of different configuration items' names in a single file into count to distinguish which file does the mapping practice. In implementation, we locate the file that do the mapping as follows: First, take the output of parsing configure file, i.e. the list of configuration items' names, as input, count the occurrences of configuration items both the kinds and times of every source file by traversing the whole source code project ; Then, rank the file by the occurrence in decreasing order, and the top one or several files would be the file contain the mapping modules.

*2) Choose the variables of each configuration items:* When found the file with mapping practice, we should analyze the file to choose the variables relevant to each configuration item. Considering the observation that the mapping practice has specific format structure and the similarity between the configuration item names and the relevant program variables, we choose textual analysis to specify the correlation bewteen configuration item name and program variables, regardless of the semantic information of the source code. The main procedure consists of four phases as follows:

*a) Dictionary establishment and weight calculation:* For each configuration item name in the list, we could divide those names into single vocabularies in terms of observations mentioned in Subsection A, thus establishing the dictionary for each configurtion option. Commonly we use uppercasing letter, underline "_", dot "." and hyphen "-" as separator. As for some regular abbreviations or computer terminologies of specific words, the relevant dictionary also adds them in. What's more, for exceptions that the configuration item names are simply connection of several words in lowercase or some regular abbreviations, a pre-build dictionary is applied to judge which word is used in the configuration item name. In addition, the order of words in the dictinary is fixed so that the chosen variable is more accurate.

After getting all the vocabularies of each configuration item, we can find out that some words appear in majority of the configuration items, such as, word "server" in configuration items of Lighttpd. It is obvious that words that appears more times in dictionaries of configuration items are less important while match the relevant program variables. Therefore, every word in those dictionaries must has a weight. In our work, the weight of each word is set as follows: First, count the occurrences of each word that appears in every dictionary of configuration item; Then calculate the reciprocal of the occurrence as its weight; Finally, for other words that doesn't occur in the dictionaries of configuration items, calculate the average weight of those existing ones as their weight.

*b) Lexemes extraction:* When we get the path of the file doing mapping practice, we match each of the configuration items in the content of that file. Once matched, the nearby lines of the matched location are going to be analyzed, and the range of those nearby lines is defined by a threshold LINE_OFFSET. For example, if LINE_OFFSET is set as 5, and we match one configuration item name at line 233, thus, the nearby lines, from line 227 to line 237, are analyzed. Further more, after adequate experiments and manul analysis, we found that the program variable mostly appears below the position of the relevant configuration items', so we could just analyze the code range from line 233 to line 237. After narrowing the analysis range down, the extraction could be more accurate and efficient.

Referring to common lexical analysis in compile principle, we simplify the traditional lexical analysis without considering the type of each token, and divide the source code into list of tokens, we name those tokens as *lexemes*. Therefore, we have a list of configuration item names with their own dictionaries and a list of lexemes for each configuration item.

*c) Relativity calculation:* For each configuration item, a lexeme should be chosen as its relevant program variables. Therefore, the relativity between each lexeme and the configuration item should be designed and calculated as a reference. In this phase, two main strategies are used to calculate the relatvity.

In our first strategy, we use some index to quantify the relativity between single word and the configuration item: 1) match number of words in the dictionary, measuring how many words in the dictionary are matched in a specific lexeme. 2) Match order of the words in the dictionary, recording the order of matched words. 3) Match ratio of the word, calculating the percentage of the matched word's bits in the total bits of both the lexeme and the configuration item. With those three quantified index, the final match relativity is calculated as follows: First, for each matched word in the dictionary, add the weight of the word in the calculation of match ratio, that is, if the word is matched sub of the lexeme, the matched bits should multiply the weight of this word while calculating the match ratio. Then, for every two matched-words, if the match order of these two words is the same with the order in the dictionary of the configuration item, the counter of right-match-order pluses by one, thus, the right-match-order ratio equals the percentage of right-match-order words in total words of the dictionary. Finally, match relativity equals the multiply of match ratio and the right-match-order ratio. To describe it more intuitive, we calculate the match relativity by the follow equation.

$$\text{Similarity}=\text{Ratio}\bullet r\text{-Ratio} \tag{1}$$

$$\text{Ratio}=\sum \text{length}(\text{word}_i)\bullet\text{weight}_i \Big/ \big(\text{length}(\text{lexeme})+\text{length}(\text{config})\big) \tag{2}$$

$$\text{r-Ratio}=\text{ro-Num}\big/\text{total-Num} \tag{3}$$

In (1), the "Similarity" means the final match relativity between the configuration item and the lexeme; the "Ratio" means the match ratio, and the "r-Ratio" means the right-match-order ratio. In (2), the function "length" return the length of the string, and the "weight$_i$" represent the weight of the "word$_i$". In (3), the "ro-Num" means the number of right-match-order word pairs, and the "total-Num" means the total number of words in the dictionary of the configuration item.

After getting the match relativity of all the lexemes, rank the lexemes in decreasing order of match relativity, then we could choose the top one as an alternated program variable.

In reality, there are some exceptions that the chosen lexeme is not the right variables, so some heuristic policies is applied to refine the results. Considering the formatted structure in mapping practice, the relative position between configuration item and relevant program variables could be some further index in relativity calculation. We define index relative line as the differential in line between configuration item and the lexeme, and index relative location as the number of words between them, to take relative position into account. In the result of the first strategy, we count the number of chosen lexemes whose match relativity is less than a predefined threshold. If this number is less than the half of the total number of lexemes, we will filter the common position that the majority of the lexemes have. Finally, we choose the lexemes with common relative position according to the relevant configuration item as the alternated program variables.

As for the second strategy, Locality Sensitive Hashing[24] (LSH) is taken into consideration. Simhash is one of the LSH algorithms to check the similarity of two texts. Once establishing the dictionary, we calculate the weight of the words in the dictionaries as well as those not. Therefore, we can calculate the simhash value of each extracted lexeme with the configuration item name. Finally, rank the lexemes in decreasing order of simhash value, and we choose the top one as another alternated program variable.

*d) Variable chosen:* After calculation of relativity between lexeme and configuration item, we can get two optional program variables for every configuration item based on two strategies mentioned above. For those configuration items that their two optional program variables are the same, we choose the lexeme as their relevant program variables. For other configuration items that are different in their two optional variables, we desert

```
/*Example 4:  httpd-2.4.12/server/core.c */
…
static const command_rec core_cmds[] = {
      …
      AP_INIT_RAW_ARGS("AccessFileName",
       set_access_name,  ⋯),
      AP_INIT_TAKE1("DocumentRoot",
      set_document_root,  ⋯),
        …
}
…
static const char *set_access_name(. . . )
{
   …
   conf->access_name = apr_pstrdup(cmd->pool, arg);
   return NULL;
}
…
static const char *set_document_root (. . . )
{
   …
   conf->ap_document_root = arg;
   return NULL;
}
…
```

Fig 4. Example of mapping function for configuration items.

the candidate variables, which ensure the accurray of chosen variables.

*3) Extract the patterns of mapping:* After variable chosen, we choose a lexeme as its relevant program variable for some of the configuration items. Then we mine the common pattern of those pair of configuration items and program variables.

First, based on the libTooling API in Clang, we generate the Abstract Syntax Tree (AST) of the file that does the mapping practice. Then, for each pair of the configuration item and program variables, we find their smallest common ancestor in the AST and record their position in the ancestor. Finally, after all the pairs find their smallest common ancestor and position, we filter the most common one as their pattern. That is, the pattern of mapping practice consists of a sub-tree in AST and the positions of the configuration item and relevant program variable in the sub-tree.

To mine the potential configuration items in the source code, we travel the AST to visit the sub-tree the same as the pattern we mined. In the iteration of the specific sub-tree, the two variables will be extracted as a pair of configuration item and relevant variables. For the sake of accuracy, we evaluate the similarity between the two members to decide whether they are a potential pair of configuration item and relevant variables.

### *D. Verify The Variables*

In the first three phases we get a relevant variable for every conifiguration option. However, it should be confirmed that these so called variables are definitely the variables in program, considering the fact that in some software we can only get the mapping function for the specific configuration rather than the direct variables, as Fig. 4 shown. In AST it is easy to figure out whether it is a variable or a function call. Then, for mapping functions, we use data flow analysis as well as the textual analysis to get the final variables that the configuration items' value assigned to. In detail, we find the implementation of the mapping functions and fetch the structure names that are assigned value to. Then, the structure definitions are analyzed to map to the specific configuration items by lexical analysis with strategies above.

## V. Evaluation

In this section, we conduct comprehensive experiments to evaluate the effectiveness of ConfMapper. We intend to answer the following research questions.

Q1: What is the accuracy of the mapping from the configuration items to relevant program variables?

Q2: What is the integrity of the mined configuration items?

Q3: What is the efficiency of ConfMapper?

### *A. Experiment Setup*

As we studied, we choose seven popular open-source software to conduct our experiments. The information of these seven software is shown in Table. I. We use the verbose configure file of the software to do the configure file parsing. In the second phase of Learning Mapping Patterns, two strategies have been applied to choose the relevant variable of a specific configuration item. So we could compare the results under different strategies to choose the most accuracy strategies and check the improvement with mapping pattern learnings. What's more, to verify the validity of our approach, we compare ConfMapper with typical keyword match. All the experiments are conducted on Dell Inspiron 3647 with Intel Core i5 4460s CPU and 12 GB memory.

TABLE I. List of Software For Experiments

| Appl. | Description | Appl. | Description |
|---|---|---|---|
| MySQL | Database | Nginx | Reverse proxy |
| Postfix | Mail proxy server | PostgreSQL | Database |
| Httpd | Webserver | Lighthttp | Webserver |
| Redis | Database | | |

```
/*Example 5: httpd-2.4.12/server/core.c*/
…
AP_INIT_TAKE1("ServerAdmin",set_server_string_slot,⋯),
…
AP_INIT_TAKE1("ErrorLog", set_server_string_slot,⋯),
…
static const char *set_server_string_slot(⋯)
{
        /* This one's pretty generic... */
        int offset = (int)(long)cmd->info;
        char *struct_ptr = (char *)cmd->server;
        …
        *(const char **)(struct_ptr + offset) = arg;
        return NULL;
}
```

Fig 5. Example of common mapping function for several configuration items

TABLE II. OVERALL RESULTS

| Appl. | Configuration Items Num.[a] | Strategy 1 | | Strategy 2 | | Strategy 1 & 2 | | mapping pattern learning | |
|---|---|---|---|---|---|---|---|---|---|
| | | *Num.*[b] | *Rate*[c] | *Num.* | *Rate* | *Num.* | *Rate* | *Num.* | *Rate* |
| MySQL | 19 | 19 | 0.53 | 19 | 0.58 | 9 | 1.0 | 322 | 1.0 |
| Postfix | 10 | 10 | 0.9 | 10 | 0.8 | 8 | 1.0 | 109 | 1.0 |
| PostgreSQL | 10 | 10 | 0.6 | 10 | 0.6 | 6 | 0.83 | 236 | 1.0 |
| Redis | 45 | 45 | 0.8 | 45 | 0.64 | 17 | 1.0 | 63 | 1.0 |
| Lighthttp | 8 | 8 | 0.5 | 8 | 0.38 | 3 | 1.0 | 43 | 1.0 |
| Httpd | 20 | 13 | 0.77 | 13 | 0.69 | 11 | 0.91 | 59 | 0.89 |
| Nginx | 9 | 9 | 0.22(0.78)[d] | 9 | 0.22(0.67)[d] | 7 | 0.22(0.67)[d] | 64 | 1.0 |

a. The number of configuration items parsed from the input configure file
b. The number of mapping in configuration items and variable pairs.
c. The correct rate of mapping in Num.
d. Some of the configuration items match the right mapping functions, but the relevant program variables are complicated structures. Only 22% of the configuration items maps to the right relevant variables.

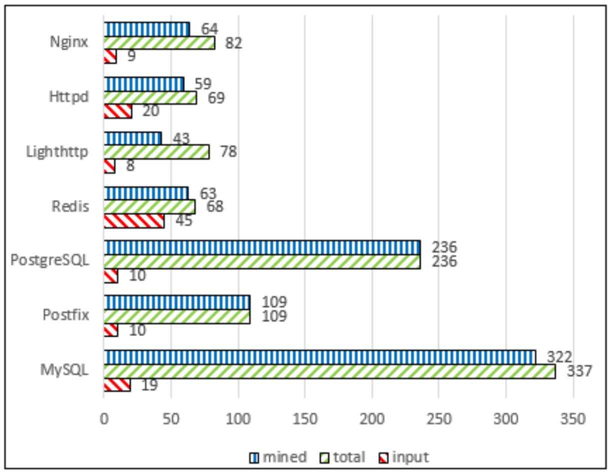


Fig 6. The numbers of configuration items for software. Index "input" means the number of configuration items we extract from the configure file; index 'total" means the total number of configuration items in the software; index "mined" means the number of configuration items we mined from the software.

## B. *Experiment Results*

In this section we display the result of our experiments and analysis of the results.

The overall results of our experiment is shown in Table II. From the Table II we can see that when we choose the variables only with strategy 1 or strategy 2, the mapping number of the configuration item and relevant variable pairs are more than that with both strategy 1 & 2, but the correct rate of these pairs are not optimistic. On the contrary, the combination of these two strategies gets fewer mapping pairs but higher correctness. With high correctness of these mapping pairs, we conduct mapping pattern learnings. As shown in Table II, the final number of mapping pairs of configuration items and relevant variables are listed in the last column. From the statistics we can figure out that the mapping accuracy of the software could reach nearly 100% except for Httpd. The reason for its inaccuracy is it use mapping function to set the configuration value to the relevant variables. While in the lexical analysis, we can only match the function name. Then there would be some deviation in the dataflow analysis considering the fact that there are some configuration items use the same mapping function to set their configure values with offset to distinguish, for instance, "ServerAdmin" and "ErrorLog" use the common mapping function "set_server_string_slot" to set their values with different "offset" in "struct_ptr", just as Fig. 5 shown.

Then we can evaluate the integrity of the configuration we mined. The numbers of configuration items in the software as well as the mined configuration items' number has shown in Fig. 6. The majority configuration items of the software are mined by ConfMapper with few input configuration items. In average, ConfMapper can mine 91.5% configuration items of the software, with three software reaches 95% or above.

What's more, we compare our method with typical keyword match in same input to evaluate the accuracy and integrity. As Fig. 7 shown, the integrity and accuracy of keyword match are much poorer than these of ConfMapper.

Finally the efficiency of ConfMapper is evaluated. We record the execution time of ConfMapper for these seven software shown in Table III. From the results we can see that

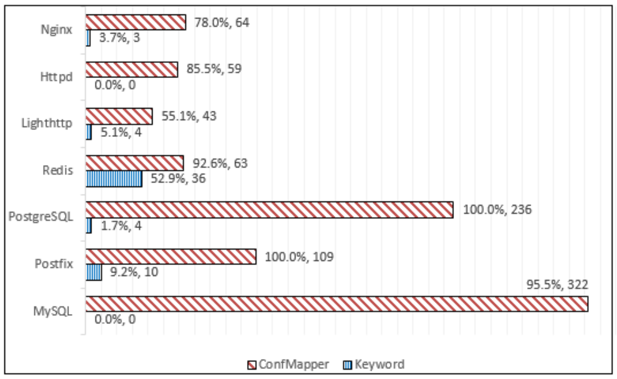


Fig 7. The comparison of ConfMapper and keyword match. The percentages in figure means the correct mapping percentages of configuration items, and the value followed means the correct mapping number of configuration items.

TABLE III. THE EXECUTION TIME OF CONFMAPPER

| Appl. | Exec. Time(second) | Appl. | Exec. Time(second) |
|---|---|---|---|
| MySQL | 3.22 | Nginx | 2.13 |
| Postfix | 18.89 | PostgreSQL | 2.96 |
| Httpd | 2.66 | Lighthttp | 0.69 |
| Redis | 1.45 | | |

the execution time are commonly less than 3.5 seconds, except for Postfix. In particular, Postfix define its configuration items' name as a list of macros in an ".h" header file, and then use these macros to map to relevant program variables So we need to preprocess the macro and header files for all the source files to find the correct mapping relation. As for Httpd and Nginx, although they use mapping functions to set the configuration values to the relevant variables, the data-flow analysis we used is simplified, so the execution time is short.

## VI. Conclusion

Misconfigurations have become the major cause of software failures. However, misconfiguration diagnosis based on program analysis face the challenges of mapping the configuration items to the relevant program variables. To fill this gap, we propose a tool named ConfMapper to automatically map the configuration items to the relevant program variables without understanding the complicated semantic context in source code. Evaluation results on seven popular open-source software demonstrate the accuracy and efficiency of ConfMapper.

## References

[1] Luiz Andre Barroso and Urs Hoelzle. 2009. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan and Claypool Publishers.

[2] Ariel Rabkin and Randy Katz. 2013. How Hadoop clusters break. IEEE Software Magazine 30, 4 (July 2013), 88–94

[3] Rabkin A. Using program analysis to reduce misconfiguration in open source systems software [Ph.D. Thesis]. Berkeley: University of California, 2012.

[4] Dong Z, Ghanavati M, Andrzejak A. Automated diagnosis of software misconfigurations based on static analysis. In Proc. of the 2013 IEEE Int'l Symp. on Software Reliability Engineering Workshops (ISSREW). 2013. [doi: 10.1109/ISSREW.2013.6688897]

[5] Attariyan M, Flinn J. Automating configuration troubleshooting with dynamic information flow analysis. In Proc. of the 9th USENIX Conf. on Operating Systems Design and Implementation (OSDI). 2010.

[6] Xu TY, Zhang JQ, Huang P, Zheng J, Sheng TW, Yuan D, Zhou YY, Pasupathy S. Do not blame users for misconfigurations. In Proc. of the 24th ACM Symp. on Operating Systems Principles (SOSP). 2013. [doi: 10.1145/2517349.2522727]

[7] Attariyan M, Chow M, Flinn J. X-Ray: Automating root-cause diagnosis of performance anomalies in production software. In Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI). Hollywood, 2012.

[8] Yuan D, Xie XL, Panigrahy R, Yang JF, Verbowski C, Kumar A. Context-Based online configuration-error detection. In: Proc. Of the 2011 USENIX Annual Technical Conf. (ATC). Portland, 2011. 313−326.

[9] Zhang JQ, Renganarayana L, Zhang XL, Ge NY, Bala V, Xu TY, Zhou YY. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In: Proc. of the 19th Int'l Conf. on Architecture Support for Programming Language and Operating Systems (ASPLOS). 2014. [doi: 10.1145/2541940.2541983]

[10] Mickens J, Szummer, Narayanan D. Snitch: Interactive decision trees for troubleshooting misconfigurations. In: Proc. of the 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SYSML). 2007.

[11] Su YY, Attariyan M, Flinn J. AutoBash: Improving configuration management with operating system causality analysis. In: Proc. of the 21st ACM Symp. on Operating Systems Principles (SOSP). 2007. [doi: 10.1145/1294261.1294284]

[12] Tucek J, Lu S, Huang C, Xanthos S, Zhou YY. Triage: Diagnosing production run failures at the user's site. In: Proc. of the 21st ACM Symp. on Operating Systems Principles (SOSP). 2007. [doi: 10.1145/1294261.1294275]

[13] Wang YM, Verbowski C, Dunagan J, Chen Y, Wang HJ, Yuan C, Zhang Z. STRIDER: A black-box, state-based approach to change and configuration management and support. In: Proc. of the 17th Large Installation Systems Administration Conf. (LISA). 2003. [doi: 10.1016/j.scico.2003.12.009]

[14] Lao N, Wen JR, Ma WY, Wang YM. Combining high level symptom descriptions and low level state information for configuration fault diagnosis. In: Proc. of the 18th Large Installation Systems Administration Conf. (LISA). 2004

[15] Wang HJ, Platt JC, Chen Y, Zhang R, Wang YM. Automatic misconfiguration troubleshooting with PeerPressure. In: Proc. of the 6th USENIX Conf. on Operating Systems Design and Implementation (OSDI). 2004.

[16] Wang HJ, Platt J, Chen Y, Zhang R, Wang YM. PeerPressure: A statistical method for automatic misconfiguration troubleshooting. Technical Report, MSR-TR-2003-80, 2003

[17] Whitaker A, Cox RS, Gribble SD. Configuration debugging as search: Finding the needle in the haystack. In: Proc. of the 6th USENIX Conf. on Operating Systems Design and Implementation (OSDI). 2004.

[18] Zhang S. ConfDiagnoser: An automated configuration error diagnosis tool for Java software. In: Proc. of the 35th Int'l Conf. on Software Engineering (ICSE). 2013. [doi: 10.1109/ICSE.2013.6606737]

[19] Zhang S, Ernst MD. Automated diagnosis of software configuration errors. In: Proc. of the 35th Int'l Conf. on Software Engineering (ICSE). 2013. [doi: 10.1109/ICSE.2013.6606577]

[20] Zhang S, Ernst MD. Which configuration item should I change? In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE). 2014. [doi: 10.1145/2568225.2568251]

[21] Clang. https://clang.llvm.org/

[22] LLVM. https://www.llvm.org/

[23] Augeas: http://augeas.net/

[24] Paulevé L, Jégou H, Amsaleg L. Locality sensitive hashing: A comparison of hash function types and querying mechanisms[J]. Pattern Recognition Letters, 2010, 31(11):1348-1358.