Automatic Type Inference for Proactive Misconfiguration Prevention

Xiangyang Xu, Shanshan Li, Yong Guo, Wei Dong, Wang Li, Xiangke Liao
College of Computer Science
National University of Defense Technology
Changsha, China
Email: {xuxiangyang11,shanshanli,yguo,wdong,liwang2015,xkliao} @nudt.edu.cn

***Abstract*—Misconfigurations have become a major cause of software failures. Most research focuses on misconfiguration diagnosis and troubleshooting, which occur after the misconfigurations have happened. Actually, if we can prevent misconfiguration before software runs, many potential catastrophic failures of systems can be avoided, thus reducing customers' downtime and support costs. In software configuration, we found that most configuration options have specific constraints, which have a strong connection with the configuration option type. If we can check the configuration settings against the inferred type before the software runs, many misconfigurations can be prevented. In this paper, we explore a name-based method called ConfTypeInferer to automatically infer the type of configuration options, which can help users to correctly configure and check settings, thus preventing misconfigurations proactively. We manually studied several popular open-source software projects to investigate the classification and naming conventions of configuration option. Based on these findings, we designed and implemented the ConfTypeInferer. We performed comprehensive experiments to evaluate the effectiveness of our method.**

***Index Terms*—misconfiguration prevention; configuration option type inference; name-based analysis;**

I. Introduction

In recent years, configuration issues have drawn tremendous attention for their increasing prevalence and severity [1][2]. One important reason for today's prevalent configuration issues is the ever-increasing complexity of configuration, which is reflected by the large and still increasing number of configuration options. For example, hundreds of configuration options need to be set for the running of database servers [3] and web servers [4]. The runtime environment of software can be determined from the settings of the configuration options, such as memory size, time interval, resource limits, etc. To set them correctly, users need to understand the meaning of each configuration option. Unfortunately, although most software manuals contain a detailed description of how to set each configuration option correctly, it is non-trivial for users to find out what the corresponding guidance is. Users rarely have the patience to look at thousands of pages of documents. They always make configurations based on experience, or exaggerate a bit, by intuition, which leads to many misconfiguration issues.

Faced with these issues, many previous studies [5] [6] [7] [8] have devoted their work to misconfiguration diagnosis and troubleshooting, which occurs after the misconfigurations have happened. Actually, if we can prevent or detect misconfigurations before software runs, many potential catastrophic failures of systems can be avoided, thus reducing not only the customers' downtime but also the support costs. Some researchers have noticed that and already conducted some work to explore misconfiguration prevention. Xu et al. [9] check parameter settings at the system's initialization time to reduce damage from failures. Encore [10] detects software misconfigurations by exploiting the interaction between the configuration settings and the executing environment, as well as the rich correlations between configuration entries. Actually, we found that most configuration options have specific constraints, which have a strong connection to the configuration option type. Fig. 1 gives two real-world examples of misconfigurations caused by an invalid type. Without knowing the option type, a user's settings often violate configuration constraints, which results in many misconfigurations. These misconfigurations waste a tremendous amount of user effort for diagnosis and might cause severe failure in the system's runtime. If we can check these configuration settings against the inferred type before the software runs, many misconfigurations can be prevented. In this paper, we try to prevent misconfiguration proactively by inferring the configuration option type for users, which can be used for checking the configuration settings, thus avoiding

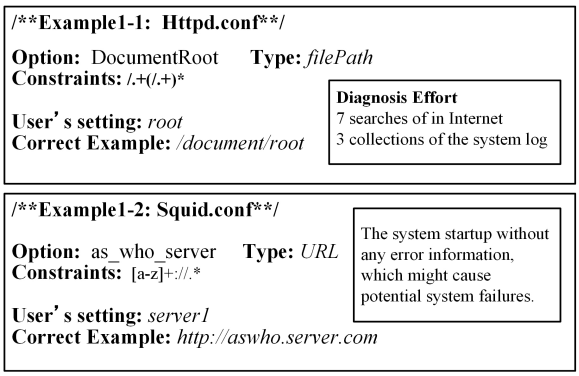

Fig. 1. Example of misconfiguration caused by invalid type. In example 1-1, the user takes a *path* type for a *filename* type, and the diagnosis cost lots of user effort. In example 1-2, the user sets a URL configuration option as a server name, which is ignored by system.

DOI reference number:10.18293/SEKE2017-072

many potential misconfigurations. We explore a name-based method called ConfTypeInferer to automatically infer the type of the configuration option.

Some challenges need to be addressed. First, we need a specific and comprehensive classification for configuration options, which is the basis of the type inference. Second, we need to figure out the naming conventions for the configuration options, which is the basis of semantic extraction. Third, we need to verify whether the configuration-option name really contains enough semantic information for type inference, or, in other words, whether the name-based method works.

To address these challenges, we manually analyzed several popular open-source software projects, such as PostgreSQL, Httpd, Nginx, Squid, etc. After a thorough study of the classification and naming conventions for their configuration-option names, we investigated the feasibility of the name-based method. Except for the *enumeration* type, the method works well for inferring most configuration option types. Then we designed and implemented ConfTypeInferer by combining name-based analysis with program analysis, in which the program analysis is to make up for the deficiency of the name-based analysis in inferring the *enumeration* type of configuration option and to verify the type inferred.

Our contributions are summarized as follows:

- Through manual analysis of several popular open-source software projects, we summarized several instances of finding configuration options, verifying the feasibility of the name-based method (Section II).
- We designed and implemented the architecture for ConfTypeInferer, the name-based method used to infer the type of configuration option (Section III and Section IV).
- We conducted comprehensive experiments to evaluate the effectiveness of ConfTypeInferer. Our results show that the accuracy of type inference can reach over 90%, and at the same time, it can prevent many misconfigurations (Section V).

II. The Feasibility of the Name-Based Method

To evaluate the feasibility of the name-based method, we try to answer the following research questions:

RQ1: How many and what types of configuration option exist in open-source software projects? Answering this question will provide a classification for type inference.

RQ2: What are the naming conventions for configuration options in open-source software projects? Answering this question will give us an in-depth understanding of configuration option names, and provide us with some ideas for mining semantic information from configuration option names.

RQ3: Do these configuration-option names convey enough information for type inference? Answering this question is the key to verify the feasibility of the name-based method.

To answer the above questions, we empirically studied more than 1,000 configuration options in several popular open-source software projects. The main findings of our study are illustrated as follows.

TABLE I
Coverage of Our Classification in Open-Source Software.

Software	Number	Coverage(%)
Redis-3.2.3	70	98.6
PostgreSQL-9.6rc1	269	98.1
Lighthttpd	273	96.2
Postfix-2.5	109	94.7
Squid-3.5.21	340	90.0
MySQL-5.7.15	732	87.7
Httpd-2.4.23	634	86.1
Nginx-1.10.2	637	85.1

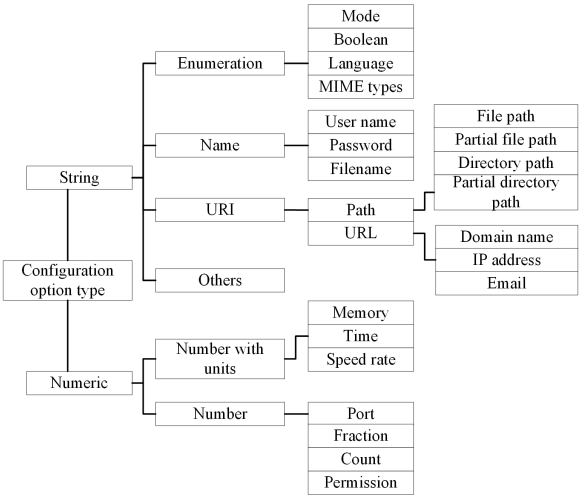

Fig. 2. Classification tree of configuration options

For RQ1, although some previous work [11] includes studies on type taxonomy, our aim is to generate configuration constraints by type inference. Therefore, we need a sufficiently fine-grained classification for all configuration options. We manually analyzed software manuals, configuration files, and even source code to classify each option. Fig. 2 illustrates our classification in the form of a tree. This classification tree can be supplemented with more software to be considered. We evaluated our classification effectiveness on about 3,000 options of eight open-source software systems by checking whether each option can be classified as one of types listed Fig. 2. As shown in Table I, the coverage of the classification is as high as 90% on average, with a minimum of 85.1%. Therefore, the result indicates the validity and efficiency of our classification.

For RQ2, we find that the configuration option names chosen by programmers are usually made up of readable words or common abbreviations connected by some separators. For example, in PostgreSQL, programmers use underscores to connect several words for the name of a configuration option, e.g., "listen_addresses," while in Httpd programmers use camel-case naming, e.g., "MaxRequestsPerChild," This naming convention makes it easier for us to perform text processing and extract semantic information. Besides, those words contained in configuration-option names usually convey explicit semantic information, including explanation, description, or constraints

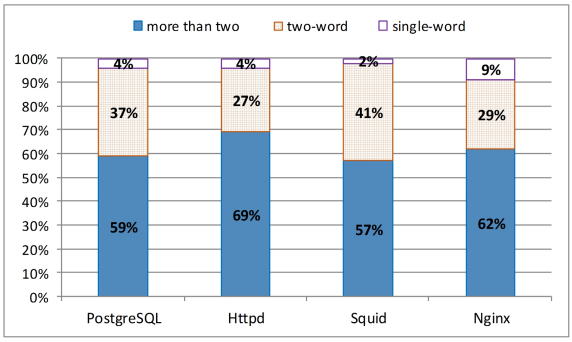

Fig. 3. The number of words in the configuration-option name.

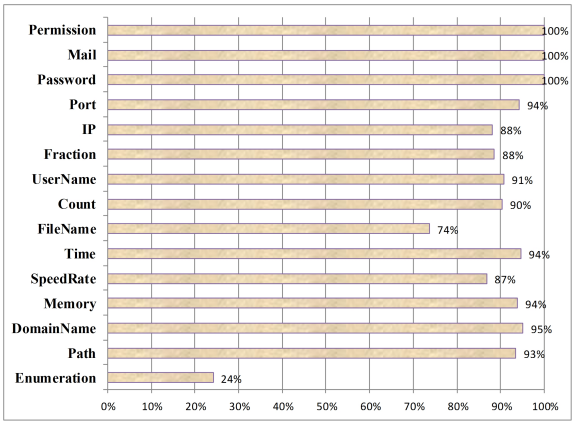

Fig. 4. The inference result of keyword matching.

about the configuration option, which reflect the option's type to some extent. For instance, the option names of *path* type usually contain keywords such as "directory," "location," "path," etc, and the option names of *memory* type usually contain keywords such as "memory," "buffer," "size," etc. These keywords can be used to build a dictionary for each configuration option type for keyword matching.

For RQ3, we studied the number of words in each configuration-option name. As Fig. 3 shows, the majority (91%-98%) of those configuration-option names contain more than one words, this observation enhances the feasibility of the name-based method because it is widely accepted that more words convey more information. In addition, we find that this name-based method doesn't work well in some cases. On the basis of the findings for RQ2, we established a dictionary for each type by collecting those high-frequency keywords in the configuration-option names, and implemented a simple program to infer the configuration option type through text processing and keyword matching. The inferring results are presented in Fig.4. We observe that the inference results are ideal (74%-100%) for most configuration option types. Unfortunately, the tool behaved poorly (only 24%) when inferring the *enumeration* type, whose ratio is relatively higher than the other type. An *enumeration* type option means the programmer enumerates all possible values to be set, while it's not impossible, but is difficult to infer the *enumeration* type from several words since there are so many words that can be used to express *enumeration* type. We chose abstract syntax tree (AST) analysis to address this problem (introduced in Section IV).

III. Architecture of ConfTypeInferer

Based on the findings in Section II, we designed and implemented ConfTypeInferer, an automatic type inferer for configuration options. Fig. 5 illustrates its architecture. In the extraction phase, we take configuration files as input, which could be the original template configuration files or user-specified ones. After parsing the configuration files through some parsing tool, we can determine the name of partial configuration options. In the mapping phase, we use the tool ConfMapper [12], which was designed to implement automatic mapping from configuration options to the relevant program variables, and we get pairs of option and variable. In the inferring phase, ConfTypeInferer conducts name-based analysis on the configuration-option names to infer the type. In addition, we identified the options for *enumeration* type by AST analysis. In the verifying phase, we conducted data-flow analysis for the program variable of each option to verify the type inferred previously.

We implemented the AST analysis and data-flow analysis on the basis of Clang [13], a C language family frontend for LLVM [14].

IV. Design and Implementation

This section presents detailed descriptions of the design and implementation of ConfTypeInferer. The goal of ConfTypeInferer is to design a user-oriented tool that automatically infers the configuration option type without any manual effort.

A. Extraction and Mapping

To infer the configuration option type, we need to extract the configuration options and corresponding variables, which is the basis of name-based analysis and program analysis. This question has been studied in our previous work [12]. We proposed a tool named ConfMapper to accomplish the automated mapping from the configuration options to the relevant program variables without needing to understand the complicated semantic context in the source code.

B. Inferring by Name-based Analysis

As discussed in Section II, the naming convention of most configuration options makes it possible for us to perform text processing and extract semantic information. However, many difficulties still exist in the implementation of name-based analysis. For instance, a keyword might reflect different configuration option types, and a name might contain several keywords that reflect different types, so how can we use the semantic information to infer the type in these cases? In ConfTypeInferer, we have come up with a scoring model to choose the most likely type. Specifically, we divide the name-based analysis into three steps. First, we separate the

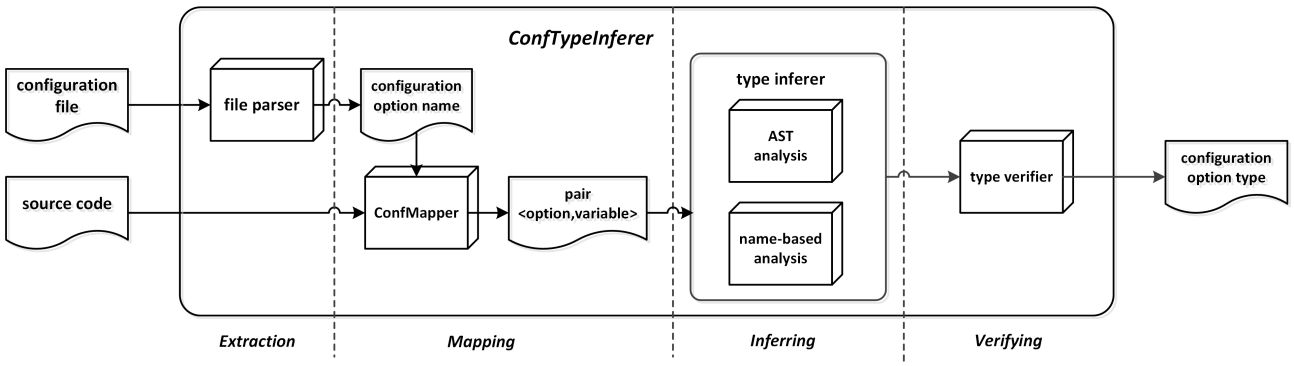

Fig. 5. The architecture of ConfTypeInferer.

configuration-option names into words by text processing, and establish a dictionary for each configuration option type, with the words ranked according to their frequency. Second, we use the scoring model to infer the configuration option type. Third, infer the *enumeration* type using AST analysis.

1) Word Segmentation and Dictionary Establishment: Based on the naming convention for the configuration option, we use uppercase letters, underscore ("_"), dot (“.”), and hyphen ("−") as separators to get words contained in the configuration-option name. We build a dictionary for each configuration option type by categorizing the words according to different types. In each dictionary, these words are ranked by their frequency of occurrence.

2) Score Model: As a name might contain several keywords, which reflect different types, and a keyword might reflect different configuration option type, the score model is described in (1).

$$score_{type} = \sum_{word_i}^{n} score_{freq} \cdot score_{order}\ 1 \le i \le n \qquad (1)$$

Here *n* means the number of keywords in an option name, $word_i$ represents the *i*th keyword, and $score_{type}$, $score_{freq}$, and $score_{order}$ mean the total score for a certain configuration option type, the frequency score for a matched word, and the order score for a matched word, respectively. The type with highest score is the inference result.

More concretely, $score_{freq}$ is set in the step for dictionary establishment, according to the word's frequency of occurrence. In our tool, we set three score levels using the function described in (2). The *x* is the frequency rank of a word.

$$score_{freq} = \begin{cases} 3 & 1 \le x \le 5 \\ 2 & 6 \le x \le 10 \\ 1 & x \ge 11 \end{cases} \qquad (2)$$

The $score_{order}$ is set according to the word's position in a configuration option name. We find that those words at the front of a configuration option name are usually used to describe the words following them, which means that the last word is most likely to reflect its type, e.g., “listen_addresses” “ssl_cert_file,” and “temp_buffers.” This is a common habit when people are naming a thing. However, there are exceptions, such as if the programmer has a strange naming habit, which would affect our inference result. In our tool, we set two score levels and, usually, the last word of a name has a higher score.

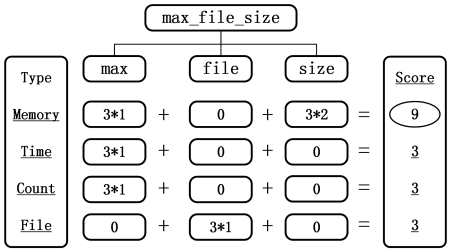

Fig. 6. An example of calculation of $score_{type}$

Fig. 6 gives an example of the calculation of $score_{type}$. The configuration-option name “max_file_size” contains three keywords that reflect different types, among which the keyword “max” could reflect further different types including memory, time, and count. After doing the calculation shown in Fig. 6, we can infer the correct type.

C. Inferring Enumeration Type

As mentioned in Section II, it is not enough to infer the configuration option type only by name-based analysis. We tried to use the program analysis method to address this issue. In order to infer the options for *enumeration* type, we manually analyzed their program variables in the source code. We find that programmers assign values to these variables by a similar pattern. In general, most software does this assignment by specific structures or functions whose name contains the configuration option or its variable, and the body of structures or functions contains many macros and strings related to the configuration option. This assignment pattern can be located and identified by AST analysis. In addition, we can also obtain all possible values of an *enumeration* type option by AST analysis.

D. Verifying the Inferred Type

We verify options' types by checking their variables' data type. This step is mainly based on our finding relating to the close connection between the configuration option type and the relevant program variables data type. For example, for a *path* type option, its variable must a string, while a *memory* type might have a variable of integer or float-point number type. In the mapping phase, we have determined the pairs of option and variable, and the data type of the variable can be obtained by a data-flow analysis.

E. Constraints Enhancement

We envisage that ConfTypeInferer is the first step towards a generic and systematic solution to prevent configuration errors. With the option type inferred, we can define constraints for a specific type, which can be used for misconfiguration checking in source code and configuration files.

1) Comments Enhancement in Configuration files: Commenting configuration files is a common practice in software development; the comments are direct, descriptive, and easy-to-understand, which can give users guidance on configuring correctly. However, we have found that the amount of useful information contained in comments is very limited in current software. With the constraints for specific option types, we can enhance comments and provide clues for user's configuration.

2) Misconfiguration Checking in Source Code: One reason for the difficulty in troubleshooting misconfigurations is the lack of diagnostic information. We can use the option type inferred to enhance the configuration constraints in the source code, which can be used for misconfiguration checking, thus improving the system's reliability and preventing many potential configuration errors.

V. Evaluation

In this section, we discuss the comprehensive experiments conducted to evaluate the effectiveness of ConfTypeInferer on two aspects: the accuracy of type inference and the effectiveness of misconfiguration prevention.

A. Accuracy of Type Inference

As we completed our study, we chose eight popular open-source software projects to evaluate the accuracy of type inference. Their information is shown in Table II. Note that there are four software projects(MySQL, Redis, Lighthttpd, and Postfix) that are not used for building dictionaries, and we chose them to verify that our name-based method works well on other C/C++ software projects. We get all the options and their correct types by manually viewing the software documentation, configuration files, and even the source code.

As Fig. 7 shows, the accuracy of type inference in open-source software projects can reach over 90%. This is an acceptable result considering the large number of configuration options. The results for MySQL, Redis, Lighthttpd, and Postfix verify that this method works well on other C/C++ software projects.

TABLE II
List of Software for Experiments

Software	Description	Software	Description
PostgreSQL	Database	Nginx	Reverse proxy
Lighthttpd	Web server	Squid	Web delivery
MySQL	Database	Redis	Database
Httpd	Web server	Postix	Mail proxy server

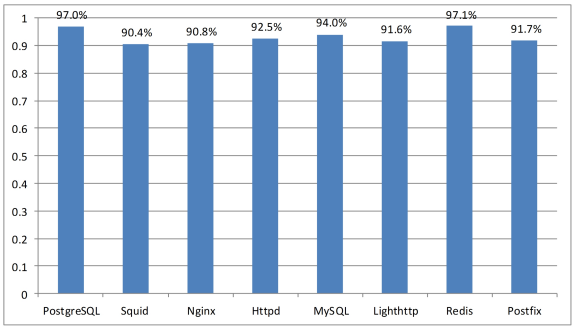

Fig. 7. Accuracy of type inference in open-source software. The percentages in the figure represent the correct inference percentages for the configuration options.

B. Effectiveness of Misconfiguration Prevention

To evaluate ConfTypeInferer's effectiveness in misconfiguraiton prevention, we performed two experiments with four software projects: PostgreSQL, Httpd, Nginx, and Postfix. In the first experiment, we injected random errors into correctly configured systems and used ConfTypeInferer to detect the injected errors. In the second experiment, we applied ConfTypeInferer to check against real-world misconfiguration problems.

1) Injected Misconfigurations: For each software project, we randomly injected 20 errors with SPEX-INJ [15] into the configuration files. SPEX-INJ automatically generates configuration errors by violating the constraints. As Table III shows, we detected most constraint violations using the option's inferred type.

2) Real-world Misconfigurations: We searched forums, frequently asked questions (FAQs) pages, and configuration documents to find actual configuration problems that users have experienced with our target software projects. In total, we chose eight representative misconfigurations to reproduce. These misconfigurations are type-related and caused by errors in the configuration files. We tried to detect these misconfigurations, given the option type inferred. Table IV lists the configuration errors for each software projects, as well as the detection

TABLE III
The number of injected misconfigurations detected by ConfTypeInferer

Software	Total	Detected
PostgreSQL	20	16
Httpd	20	13
Nginx	20	12
Postfix	20	15

TABLE IV
DETECTION OF REAL-WORLD MISCONFIGURATIONS

ID	Software	Problem Description	Success
1	PostgreSQL	Logging is not performed because log_directory (path) is set incorrectly	Y
2	PostgreSQL	Query operation is very slow due do the work_mem (memory) option being set too low	N
3	Httpd	Website visitors are unable to upload files due to the wrong permission (permission) being set	N
4	Httpd	Unable resolve PHP code due to setting the AddType (enumeration) option as a freedom string	Y
5	Nginx	File creation error due to datadir's wrong owner (username)	Y
6	Nginx	Failed to connect to the proxy server due to the wrong proxy_pass (url) being set	Y
7	Postfix	Cannot deliver mail locally due to the mydestination (email) option being set incorrectly	Y
8	Postfix	Cannot forward user's email to the Internet due to the relayhost (email) option is set incorrectly	Y

results. Table IV shows many (6/8) misconfigurations can be prevented by type checking. However, some misconfigurations cannot be prevented even when the type is inferred. For example, for Problem ID 3, you need more detailed constraints to set a *memory* option. This limit has inspired our future work for inferring more detailed constraints based on the inferred type.

VI. RELATED WORK

To prevent misconfigurations, some research detects misconfigurations by constraint verification. ECC Fixer [16] infers configuration constraints by program analysis and detects the configuration violations, it designs an algorithm that automatically generates range fixes for a violated constraint. SPEX [15] infers configuration constraints from source codes, and use these constraints to harden systems against configuration errors and to detect error-prone designs. These methods all neglect the semantic information of configuration-option names, which can complement program analysis.

The identifier names chosen by developers convey information about the semantics of a program. This information can complement traditional program analyses in various software engineering tasks, such as bug finding, code completion, and documentation. Recent work uses identifier names to infer API specifications [17], to synthesize code completions [18], and to detect incorrectly ordered method arguments of the same type [19].

VII. CONCLUSION

Misconfigurations have become a major cause of software failures. In this paper, we have explored a name-based method called ConfTypeInferer to automatically infer the type of a configuration option, which can help users to configure correctly and check their settings, avoiding many unnecessary misconfigurations. We manually studied several popular open-source software projects and research on the classification and naming conventions for configuration option. Based on these findings, we designed and implemented the ConfTypeInferer. We conducted comprehensive experiments to evaluate the effectiveness of ConfTypeInferer.

ACKNOWLEDGMENT

This paper is partially supported by NSFC No. 61532007, No. 61690203, and No. 61402496.

REFERENCES

[1] L. Barroso, J. Clidaras, and U. Hoelzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," vol. 8, no. 3, p. 154, 2009.
[2] A. Rabkin and R. Katz, "How hadoop clusters break," *IEEE Software*, vol. 30, no. 4, pp. 88–94, 2013.
[3] MySQL, http://www.mysql.com/.
[4] Httpd, http://httpd.apache.org/.
[5] Y. Y. Su, M. Attariyan, and J. Flinn, "Autobash: improving configuration management with operating system causality analysis." in *ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, Usa, October*, 2007, pp. 362–371.
[6] J. Mickens, M. Szummer, and D. Narayanan, "Snitch: interactive decision trees for troubleshooting misconfigurations," in *Usenix Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, 2007, p. 8.
[7] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *Usenix Conference on Operating Systems Design and Implementation*, 2010, pp. 1–11.
[8] Z. Dong, M. Ghanavati, and A. Andrzejak, "Automated diagnosis of software misconfigurations based on static analysis," in *IEEE International Symposium on Software Reliability Engineering Workshops*, 2013, pp. 162–168.
[9] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 619–634. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026925
[10] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, *EnCore: exploiting system environment and correlation information for misconfiguration detection*. ACM, 2014.
[11] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *International Conference on Software Engineering*, 2011, pp. 131–140.
[12] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, "Confmapper: automated variable finding for configuration items in source code," in *International Conference on Software Quality, Reliability and Security-Companion*, 2016.
[13] Clang, https://clang.llvm.org/.
[14] LLVM, https://www.llvm.org/.
[15] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 244–259.
[16] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *International Conference on Software Engineering*, 2012, pp. 58–68.
[17] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *International Conference on Software Engineering*, 2012, pp. 815–825.
[18] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 419–428, 2014.
[19] M. Pradel and T. R. Gross, "Name-based analysis of equally typed method arguments," *IEEE Transactions on Software Engineering*, vol. 39, no. 39, pp. 1127–1143, 2013.