Dissertation
submitted to the
Combined Faculties for the Natural Sciences and for Mathematics
of the Ruperto-Carola University of Heidelberg, Germany
for the degree of
Doctor of Natural Sciences

presented by

Diplom-.....

born in:.....

Oral-examination:.....

Diagnosing Software Configuration Errors via Static Analysis

Adviser: Prof. Dr. Artur Andrzejak

Abstract

Software misconfiguration is responsible for a substantial part of today's system failures, causing about one quarter of all user-reported issues. Identifying their root causes can be costly in terms of time and human resources. To reduce the effort, researchers from industry and academia have developed many techniques to assist software engineers in troubleshooting software configuration.

Unfortunately, there exist some challenges in applying these techniques to diagnose software misconfigurations considering that data or operations they require are difficult to achieve in practice. For instance, some techniques rely on a data base of configuration data, which is often not publicly available for reasons of data privacy. Some techniques heavily rely on runtime information of a failure run, which requires to reproduce a configuration error and rerun misconfigured systems. Reproducing a configuration error is costly since misconfiguration is highly relevant to operating environment. Some other techniques need testing oracles, which challenges ordinary end users.

This thesis explores techniques for diagnosing configuration errors which can be deployed in practice. We develop techniques for troubleshooting software configuration, which rely on static analysis of a software system and do not need to execute the application. The source code and configuration documents of a system required by the techniques are often available, especially for open source software programs. Our techniques can be deployed as third-party services.

The first technique addresses configuration errors due to erroneous option values. Our technique analyzes software programs and infer whether there exists an possible execution path from where an option value is loaded to the code location where the failure becomes visible. Options whose values might flow into such a crashing site are considered possible root causes of the error. Finally, we compute the correlation degrees of these options with the error using stack traces information of the error and rank them. The top-ranked options are more likely to be the root cause of the error. Our evaluation shows the technique is highly effective in diagnosing the root causes of configuration errors.

The second technique automatically extracts names of options read by a program and their read points in the source code. We first identify statements loading option values, then infer which options are read by each statement, and finally output a map of these options and their read points. With the map, we are able to detect options in the documents which are not read by the corresponding version of the program. This allows locating configuration errors due to inconsistencies between configuration documents and source code. Our evaluation shows that the technique can precisely identify option read points and infer option names, and discovers multiple previously unknown inconsistencies between documented options and source code.

Zusammenfassung

Konfigurationsfehler sind für einen erheblichen Teil heutiger Systemfehler verantwortlich und verursachen etwa ein Viertel aller von Nutzern gemeldeten Probleme. Ihre Ursachen zu bestimmen kann sehr teuer sein, gemessen in Zeit und menschlichen Ressourcen. Um den Aufwand zu reduzieren, haben Forscher aus der Industrie und Akademia viele Techniken entwickelt, um den Software-Ingenieuren beim Beheben von Problemen mit Softwarekonfigurationen zu helfen.

Dennoch bleibt es eine Herausforderung, diese Techniken anzuwenden, um Konfigurationsfehler zu erkennen, da die Daten und Eingriffe, die für deren Anwendung notwendig sind, in der Praxis schwer zu beschaffen bzw. durchzuführen sind. Zum Beispiel benötigen manche Techniken Datenbanken mit Konfigurationsdaten, die aufgrund von Datenschutzbedenken oftmals nicht frei verfügbar sind. Manche Techniken hängen stark von Laufzeitinformation fehlgeschlagener Programmläufe ab, was deren Reproduktion durch wiederholtes Ausführen nötig macht. Das Reproduzieren von Fehlern durch Miskonfiguration ist teuer, da Miskonfigurationen stark von der Umgebung bei der Ausführung abhängen. Manche anderen Techniken verwenden Test-Orakel, deren Einsatz gewöhnliche Nutzer überfordert.

Diese Arbeit untersucht praktisch anwendbare Techniken zur Diagnose von Fehlern durch Miskonfigurationen. In dieser Dissertation wurden Techniken zur Diagnose von Konfigurationsfehler entwickelt, die auf statischer Analyse von Software-Systemen beruhen und keine Ausführung des Systems benötigen. Unsere Techniken setzen nur voraus, dass der Quelltext der Anwendung und die Dokumentation der Konfigurationen verfügbar sind. Dies ist heutzutage oft der Fall, insbesondere im Fall von Open-Source-Software. Unsere Techniken können als Dienste angeboten werden.

Die erste Technik behandelt Fehler durch fehlerhafte Werte von Konfigurationsoptionen. Sie analysiert Softwareprogramme und errechnet, ob ein möglicher Ausführungspfad besteht zwischen den Orten, an denen der Wert der Option eingelesen wird, und dem Ort des Auftretens des Programmabsturzes bei der Ausführung. Konfigurationsoptionen, deren Wert Einfluss auf diesen Absturzort haben könnten, werden als mögliche Ursachen in Betracht gezogen. Schließlich werden die Korrelationsgrade dieser Optionen mit den durch die Stacktraces aufgezeichneten Absturzorten ermittelt und die Optionen danach in eine Rangfolge gebracht. Die ranghöchsten Optionen haben die höchste Wahrscheinlichkeit, eine Ursache des Absturzes zu sein. Unsere Auswertung zeigt, dass diese Technik sehr effektiv dabei ist, die Ursache von Miskonfigurationsfehlern zu identifizieren.

Die zweite Technik extrahiert Konfigurationsoptionen, die ein Programm einliest, inklusive der Orte im Programmquellcode, an denen diese eingelesen werden. Wir identifizieren zunächst Anweisungen, die Konfigurationsoptionen laden, ermitteln dann, welche Optionen durch diese Anweisungen gelesen werden, und geben schließlich eine Zuordnung der Optionen zu den Einleseorten aus. Diese Zuordnung erlaubt uns,

Optionen zu finden, die dokumentiert sind, aber tatsächlich nicht vom Programm der zugehörigen Version eingelesen werden. So können Konfigurationsfehler vermieden werden, die auf Inkonsistenz zwischen Dokumentation der Konfigurationsoptionen und dem Quellcode beruhen. Unsere Auswertung zeigt, dass diese Technik Einleseorte von Optionen präzise bestimmen und Optionsnamen erkennen kann. Sie konnte auch mehrere zuvor unbekannte Inkonsistenzen zwischen Dokumentation von Konfigurationen und Quellcode in einer großen Anwendung aufzeigen.

Acknowledgements

I owe deep thanks to my advisor Artur Andrzejak, for giving me the chance to step into the world of software engineering. I am profoundly grateful to Artur, for giving me the freedom to explore and trusting me to tackle real-world software issues. He has taught me a great deal over last 4 years in research, working on real world problems, developing practical solutions, and being simple and specific. His tireless feedback on ideas, paper drafts, and talks vastly increase the quality of my work. Re-reading my old papers, I am struck how very much his advice has improved my scientific writing. His many suggestions on life and career will influence me in my future pursuits.

Thanks to all my colleagues and friends, Mohammadreza Ghanavati, Lutz Büch, Diego Costa, and Kai Chen. Over years, they have been so friendly and patient to give advice on my projects and review my manuscripts over and over again before deadlines. Of course, thanks to them for all the fun time and discussion we had. Special thanks to Lutz Büch for the help in my life in Germany. I cannot speak German and have got a lot of help from him. I cannot remember how many letters he has read and how many calls he has made for me. Without his help, I would have had a lot of troubles.

Thanks to the institute of Computer Science, Heidelberg University for providing a great research environment. The two rows of cherry trees near my old office (Im Neuenheimer Feld 348), so beautiful for all seasons, had accompanied me for 4 years and made my life so enjoyable.

To my wife and parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays most software systems are designed to be configurable for handling variations in user and target-platform requirements. By configuration setting, various instances of a configurable software system can be generated for different scenarios and use cases. The preferred method of providing configurability for software programs is to expose a series of *configuration options*, also known as *parameters*. Specifying different values for a configuration option can activate or inactivate the software functionality associated with the option. This brings a lot of convenience for users to customize software systems based on their requirements.

Due to the growing functionality and size of today's software, the number of configuration options in a software system increases at a rapid rate. The number of configuration options in a highly-configurable system can reach thousands. For instance, Mozilla Firefox 43.0, an open source web browser, has over 2000 configuration options available to users [1]. Apache Hadoop 2.7.1., a software framework, has more than 800 configuration options [34]. MySQL database server 5.6 has more than 461 configuration options controlling different features and adjusting the performance at runtime [5]. Apache HTTP server 2.4 has 550+ configuration options across all modules [2].

1.1 Configurable Software Maintenance

Configurable software systems require costly maintenance. Meanwhile, mistakes can be easily introduced from users during configuration setting and lead to system failures. Diagnosing these failures takes a huge amount of human effort. On the other

hand, additional effort is needed to document configuration options and maintain them as configurable software evolves. We further discuss it in these two aspects.

1.1.1 Software Misconfiguration

Configuration options bring much flexibility in customizing an application. On the other hand, configuration options give users choices to determine option values. Users can easily make mistakes in configuration setting due to the following factors.

First, little guidance is provided for configuration setting. The survey by Hubaux et al. shows 56% of Linux and eCos users complained about the lack of guidance for making configuration decisions [38]. Documents searched over the Internet are often out of date, i.e., do not match with the system a user is working on [46].

Second, users are typically not well trained. Misunderstanding documents or ignorance of instructions introduces errors in configuration setting. For instance, Rabkin and Katz report that memory mismanagement causes approximately one-third of all misconfiguration problems in Hadoop systems [62]. The major reason is that users do not have a deep understanding of how Hadoop manages the memory.

Finally, the design and implementation of configuration mechanisms are not user-friendly. Despite the emergence of configuration wizards and graphical user interfaces, configuration files still represent the primary means of storing option data. Errors can be easily introduced in the configuration file due to typos, structural mistakes and semantic mistakes. These errors are often difficult to be realized by a user without the aid of functionality that validates option values in a system.

Due to these factors, misconfigurations frequently occur in the administration of software systems. Studies show software misconfigurations are one of the major causes of today's system failures [32, 53, 54, 87, 59]. The investigation in [88] shows that around 27% of the issues in one company's costumer-support database are labeled as configuration-related.

Diagnosing configuration errors is difficult for end users. Diagnostic messages in a configuration error are often inadequate and do not provide a clear hint for solving the error. According to studies [38, 88], a diagnostic message is often cryptic, hard to understand, or even misleading. Another investigation [13] also indicates that up to 25% of a software maintainer's time is spent on following blind alleys suggested by poorly constructed and unclear messages. Users cannot easily solve configuration errors with limited domain knowledge.

Eventually, a large number of configuration errors are reported to developers or technical support. Remotely diagnosing these configuration errors for users requires tremendous efforts, which significantly increases the cost of software administration [43].

1.1.2 Configuration Options Documentation

Software projects often adopt a distributed configuration management model, which does not have a centralized module for dealing with all configuration data in a software system [60]. Instead, each module in the system has its own configuration file for users setting option values and loads the file at runtime. If a predefined option is given a value in the file, the value will be loaded and create the corresponding effect on the system at runtime. Otherwise, the default value of this option will be used.

This configuration management model allows developers to freely add, remove, and modify options without accessing any global module in the software development. But it brings challenges for configuration option documentation.

Documenting configuration options requires a huge amount of manual effort. Documentation is often separated from code development. Without a centralized option management module, option documentation for a software system needs communication with developers of each module. Otherwise, a documentation writer has to manually inspect code and extract options based on convention, which is time-consuming and requires a deep understanding of how options are used in the program.

Besides, configuration options change as software evolves [93]. For instance, the number of configuration options in Apache Hadoop [34] increased to over 800 in version 2.7.1 from 141 in version 0.20.2. Maintaining configuration options requires human effort as well.

In practice, human resource for documentation is limited in a project. Documentation only comes after a major release and cannot stay in step with code evolution. This lag can lead to inconsistencies between documentation and source code, i.e., option changes in source code are not timely updated in documentation as software evolves. These inconsistencies can bring tricky misconfiguration issues. For instance, in the guidance of documentation, users could use a removed option in the system, which brings frustrating experience that the setting stated in documentation cannot create expected behavior on the system.

1.2 Our Approaches for Troubleshooting Software Configuration

Automated diagnosis of such configuration errors can significantly decrease the cost of maintaining configurable software systems and has been attracting a lot of attention from academia and industry [61, 91, 87, 52, 85, 89, 60, 12, 10, 84, 72, 76, 90, 28, 63, 35, 49, 80]. Many techniques in prior work have a good performance in diagnosing configuration errors, i.e., precisely pinpointing root causes of software misconfigurations. However, applying those techniques in practice still faces challenges such as collecting user configuration data, changing the executing environment, heavily relying on run time information. We will discuss those approaches and their challenges in Chapter 3.

In this thesis, we explore using static program analysis techniques to develop approaches of diagnosing configuration errors. The proposed approaches can be deployed in practice without challenges previous techniques have. Our work mainly consists of two parts. First, we develop an approach capable of diagnosing configuration errors due to mistakes in setting configuration values. Second, we propose an approach for identifying statements loading configuration option values in the source code, which can be used to diagnose configuration errors due to inconsistencies between source code and documentation. Both approaches are implemented as tools, namely *ConfDoctor* and *ORPLocator*. Next, we briefly introduce the two approaches.

1.2.1 ConfDoctor: Automated Diagnosis of Configuration Errors

Parameter-related misconfigurations are a major part of user configuration errors according to studies [88, 60, 9]. These configuration errors are mainly caused by mistakes from users at setting option values. Pinpointing the incorrectly-configured option from hundreds, even thousands of options in a system is difficult for a user with limited domain knowledge.

ConfDoctor addresses this types of configuration errors. The core idea of ConfDoctor is to use static analysis to track down data flow of each option value in the program and infer whether the data flow goes to the crashing site of an error. Options whose data flow goes to the crashing site are considered relevant to the error. Then we further compute correlation degrees of each relevant option with the error

by analyzing how possible its data flow occurs prior to a failure. Finally ConfDoctor outputs a ranked list of options based on their correlation degrees. Options at the top of the list are considered as potential root causes of the error.

Relying on static analysis, ConfDoctor requires the program of a being diagnosed system, all option names of the system (not option values), and the stack traces of a configuration error. The program and option names of a system are often publicly available. Stack traces are assumed in the error message. ConfDoctor can be deployed as a third-party service for diagnosing configuration errors since our analysis is fully automated. Compared to prior work, ConfDoctor has advantages in several aspects.

End-users Oriented. ConfDoctor can be deployed as a service to diagnose configuration errors for users as we stated. Users do not need to report them to developers. This solves two major problems of diagnosing configuration errors at developers site.

First, reproducing errors is needed at developers or technical support site. This can be hard and costly. In the real world, reproducing classical (i.e. code) errors is quite difficult. Errors often occur during production runs. Users prefer not to report all essential information to reproduce an error because of privacy and economic concerns. Research [67] has shown that there is a strong mismatch between what developers need to reproduce and fix a bug and what users tend to provide. Another study [16] has shown that bug reports lack information needed for bug reproduction. For some cases, reproducing configuration errors can be more costly and critical in terms of data privacy than reproducing classical errors. One reason is that a misconfiguration might manifest only with specific settings or a state of the runtime environment. This environment information needs to be collected for error reproduction, and the environment needs to be replicated as well.

Second, users cannot get a response from developers in short time. Research [87] finds that developers take laid-back roles in handling misconfiguration problems. Configuration errors are considered user's faults instead of bugs in software. It takes a long time getting answers from developers for configuration errors.

Reducing Necessity for Runtime Information. Our diagnosis targets crashing failures due to misconfiguration. The main required information from the failure run of a system is stack traces. There is no need to for program re-execution, code instrumentation such as profiling information during runtime. Reproducing errors is not needed.

Requiring No Sensitive Data from Users. The diagnosis does not require configuration data of a user and information of runtime environment. The program of being analyzed and its list of configuration options required by our analysis are often publicly available, especially for open source software.

1.2.2 ORPLocator: Identifying Read Points of Configuration Options

Configuration options documentation requires tremendous human effort as we stated. The major part of the effort is spent on acquiring where options are used in code, ensuring documented options are used and used options are documented. Automatically identifying where options are used in source code would significantly reduce human effort in maintaining configuration options documentation.

ORPLocator addresses this issue by identifying where an option value is loaded in the source code and guaranteeing consistency of options between source code and documentation. Given a program, ORPLocator is capable of extracting all options read in the program and identifying where these options are read. One can easily finds inconsistent options by comparing extracted options with documented options.

The core idea of ORPLocator is to mark the methods of loading configuration option values, locate the call sites of these methods by analyzing the program, finally infer which option is read at each call site. Then a map is built between names of options read by the program and their locations of being read in the source code. ORPlocator can significantly reduce human effort in multiple aspects for the maintenance of configurable software systems.

Configuration Options Documentation. ORPLocator can assist a documentation writer by producing a first draft of configuration options documentation. More importantly, it can extract configuration options for a subsequent version and easily identify changes of configuration options. This saves the effort of acquiring options changes during software evolution by manual code inspection or communication with related developers.

Options Inconsistencies Detection. As we stated above, inconsistencies between documentation and source code can lead to tricky software misconfigurations. ORPLocator can help detect these inconsistent options by extracting configuration options in the present version of a program. Ineffective options for the version can be detected by comparing extracted options against documentation. Analogously, it also

effectively detects inconsistent options between configuration files and source code of a program.

Automated Misconfiguration Diagnosis. A large body of research [60, 87, 92, 12, 91, 25, 26] has attempted to automatically diagnose software configuration errors. The approaches here include tracing the data flow of option values using program analysis techniques such as program slicing and taint analysis. A prerequisite for almost all of these works is a list of (typically manually specified) option read points. Thus, our approach and tool can further automate these approaches and save considerable manual efforts.

Extraction of Configuration Option Constraints. Another branch of work [87, 91, 51, 52] attempts to prevent configuration errors by telling users whether given option values violate pre-defined rules or a set of constraints. Also in this case the identification of option read points is a requirement for using such techniques.

1.3 Design Principles

The large amount of configuration options in highly configurable systems is one of major challenges for diagnosing a configuration error and makes it difficult to manually check the values of all options. Those highly configurable software systems often have large scales. This thesis attempts to develop practical techniques for solving these issues. Our techniques should scale to large software systems and can be deployed in practice.

Embracing imprecise Program Analysis. Program analysis has been developed for several decades and can apply to industrial-level software systems. But it is still challenging to apply program analysis to large-scale software systems. In order to analyze large-scale software systems, we have to make tradeoffs between scalability and precision of program analysis techniques.

We embrace the imprecise program analysis on large-scale software systems. Sound analysis is expensive and not feasible for large-scale software systems. We give up analyzing certain behavior of systems on a fine-grained level. For instance, heap dependence has to be ignored in the analysis of diagnosing configuration errors. Control dependence is not considered for some analyses as well. Our goal is to sufficiently solve targeted problems based on feasible program analysis, i.e., diagnosing the root cause of a configuration error. Guaranteeing theoretical soundness and completeness of program analysis is not our priority.

Our evaluation does not rely on formal proofs. We conduct experiments to demonstrate that our techniques are sufficiently effective. To make our empirical claims convincing, we used 29 configuration errors across 4 subject applications to validate ConfDoctor's effectiveness. For the evaluation of ORPLocator, we chose the latest version of Apache Hadoop (version 2.7.1 before Jan. 2016) as our subject application, which scales to over 1 million lines of code.

Being Deployable. Our analysis techniques target end users and cannot rely on some assumptions which are difficult to achieve in practice. For instance, users are willing to share configuration data and parameters of runtime environment for diagnosing misconfiguration. Users accept versions of a program with significantly low performance in practice, i.e., code-instrumented version of an application for obtaining behavior data at runtime. Or users have enough domain knowledge to get testing oracles when errors occur. Our analysis should not base this type of assumptions.

Our analyses require source code being analyzed programs and its list of configuration options. We argue this requirement is reasonable in the age of open-source software flourishing. In addition, the diagnosis of configuration errors needs the information of stack traces of a crashing failure, which is available in the occurrence of a crashing error.

A major cause of static analysis techniques not being widely used in practice is the high rate of false-positives in analysis results [56]. Our analyses needs to achieve high accuracy and low false-positives in solving targeted problems. In our approaches, we develop models and adopt heuristics to achieve high accuracy and precision of diagnosis results.

1.4 Contributions

The thesis makes the following contributions in the field of software engineering. The unique contributions are as follows:

- We propose an automated approach for diagnosing configuration errors which relies on only static analysis. Compared to prior work, the approach does not require reproducing errors and OS-level support. After a one-time preprocessing of the source code and configuration options, it needs as only runtime input the stack trace of the current error. It can be deployed as a third-party service for end users.

- We develop a model to compute the correlation degree of each option with a configuration error. Addressing the imprecise results from static analysis techniques, a traditional problem for static analysis, we propose multiple heuristics to improve the model of computing correlation degrees and make our approach capable of precisely pinpointing the root cause of a configuration error.

- We implement the proposed approach, ConfDoctor, and evaluate the accuracy of the approach on 29 configuration errors from 4 open source application programs - JChord, Randoop, Hadoop, and HBase. ConfDoctor can successfully diagnose 27 out of 29 errors. For 20 errors, the root cause configuration option is the first-ranked suggestion. For the other 7 diagnosed errors, the root cause is ranked in the top four.

- We develop an approach for automatically extracting names of options read in a program and locating where these options are read. Different from prior work, our approach only scans source code of an application program and does not have the issue that partial code is not analyzed due to imprecise reflection analysis. ORPLocator can achieve more precise results than prior approaches.

- We implement our approach, ORPLocator, and conduct an empirical study on the latest version (2.7.1) of Apache Hadoop, a widely popular framework for distributed data processing with more than 1.3 million lines of source code and 800+ configuration options. Results show that our approach is effective in identifying option read points in source code. Besides, our study discovers multiple previously unknown inconsistencies between documented options and source code.

In summary, this thesis takes a solid step towards automated diagnosis of software configuration issues. Our techniques are able to diagnose root causes of configuration errors in a fully automated way and can be deployed in practice without obstacles which existing techniques face. Our approaches and implementations incorporate empirical observations on usage of configuration options for diagnosing software misconfigurations. This significantly improves the scalability of the analysis on large-scale software systems. Our techniques can facilitate and enhance a variety of research efforts which require tracing configuration data in the software systems, e.g., extraction of configuration rules and validation of software configuration.

Our work has been published in top-ranked venues in the domain of software engineering.

- Zhen Dong, Artur Andrzejak, David Lo, and Diego Elias Costa. Orplocator: Identifying reading points of configuration options via static analysis. In *IEEE 27th International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, Canada, 23-27 October*, 2016
- Zhen Dong, Artur Andrzejak, and Kun Shao. Practical and accurate pinpointing of configuration errors using static analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 171–180, 2015
- Zhen Dong, Mohammadreza Ghanavati, and Artur Andrzejak. Automated diagnosis of software misconfigurations based on static analysis. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013 - Supplemental Proceedings*, pages 162–168, 2013

The following publication is related to a broader field of automated software debugging.

- Mohammadreza Ghanavati, Artur Andrzejak, and Zhen Dong. Scalable isolation of failure-inducing changes via version comparison. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013 - Supplemental Proceedings*, pages 150–156, 2013

1.5 Outline

This thesis is organized into six chapters. The next Chapter 2 introduces foundations of our work. First, we present a configurable software system architecture and common implementation mechanisms. Second, we introduce concepts of static analysis techniques used in the thesis and briefly explain how they work.

Chapter 3 presents related work. In this chapter, we summarize prior techniques of software configuration troubleshooting and group them into two parts: misconfiguration prevention and misconfiguration diagnosis. Then we briefly introduce how these techniques work and discuss their limitations. The purpose of summaries is to provide the background of our work.

Chapter 4 presents the approach of automated diagnosis of software configuration errors. In this chapter, we describe addressed configuration errors, explain details of our approach, and present its evaluation. Finally, we discuss its limitations and threats to validity.

Chapter 5 presents our work of automatically identifying option read points for configurable software systems. We first describe addressed configuration issues, then explains how our approach works, and present its evaluation. Finally, we discuss its limitations and threats to validity.

Chapter 6 presents conclusions drawn from our work and discusses possible directions for future work.

Chapter 2

Foundations

This chapter introduces configurable software systems and static analysis techniques we used in this thesis. For configurable systems, we focus on the configuration mechanism implementations in today's software systems and provide the context of configuration issues we work on. For static analysis techniques, we mainly introduce concepts and briefly explain how these techniques work, which helps understand our approaches.

2.1 Configurable Software Systems

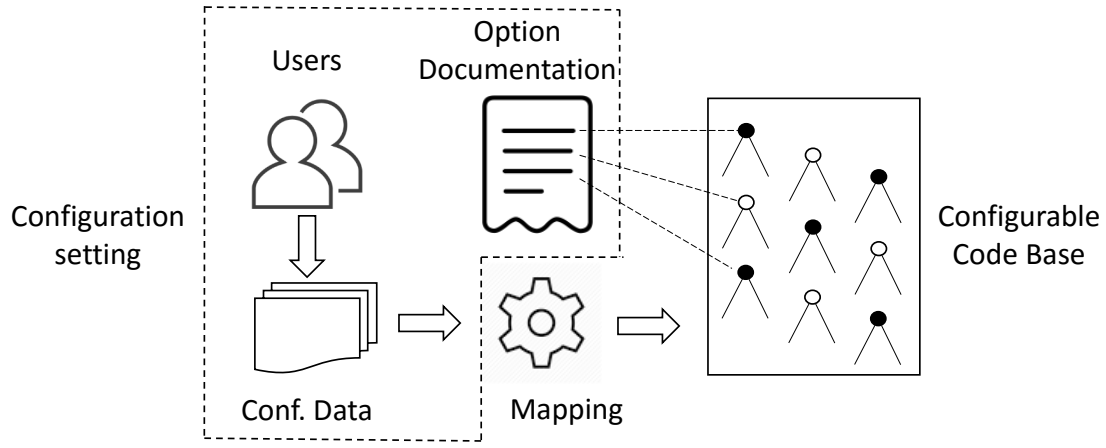

Figure 2.1: The configuration mechanism in a configurable system

A configurable software system is a configurable code base and a set of mechanisms for implementing predefined variations in the system's structure and behavior [65, 22].

These variations can satisfy customer individuality requirements and deal with the diversity of computing platforms, e.g., Windows and Linux.

Figure 2.1 shows the configuration mechanism in a configurable system, which mainly is composed by three components: configuration setting, mapping, and a configurable code base. The configuration setting component allows a user to specify different values for configuration options. The specified option values, also called *configuration data*, are put into predefined places. Then mapping component is in charge of loading these option values by dedicated interfaces in the system and mapping them into the configurable code base. The configurable code base generates a specific variant of the system based on option values, which runs as required by a user.

Next, we further present the implementations of each component in the configuration mechanism. We first introduce configurable code base, then configuration setting and mapping.

2.1.1 Configurable Code Base

The configurable code base consists of all code of functional modules as designed and implemented in a system. For configurability, there exist many preset points in code base, which allow multiple changes. These variations can inactivate or activate code of a fragment, component, module, or library, making the system run differently. Each preset point normally corresponds to a *knob* which controls all variations of the preset point. The knob is called a *configuration option*, also known as *a parameter, a preference* . These configuration options and their usages are typically documented and released to users. With guidance of the documentation, a user can set different values for configuration options as needed and make the system run as required.

There are many techniques for making code changeable or configurable. Among all techniques, *annotative approaches* and *compositional approaches* are typically used in modern software systems. Here we briefly introduce how they work in configurable software systems.

Annotative Approaches

Software variability in annotative approaches comes from the selection of annotated code fragments at compile time. A code base is composed of common code and variable code, which are not separated and mixed together. Common code is shared

by all instances of a configurable software system. Variable code is annotated code with presence conditions. A presence condition is a propositional formula representing a set of valid configurations. Given a configuration, some annotated code fragments can be included or excluded based on the evaluated results of presence conditions, which forms individual variants of the configurable software system.

This kind of configuration options can be defined using #define macro in the source code directly, or externally in MAKE-FILEs, configuration files, configuration tools, or in the form of compiler parameters. Developers can use tools to generate complex configuration setting by combining configuration options using different operators such as logical reasoning and bit manipulations.

We use an example to illustrate the work mechanism of annotative approaches. Figure 2.2 shows an implementation of a bicycle. The method *drive* is common code including lines 7 to 10. All code between keywords *#ifdef* and *#endif* is annotated code. The annotated code is controlled by *#ifdef* conditions in lines 3 and 12. If the configuration option *Motor* is specified, the bicycle will equipped with an engine and become a motorbicycle.

Compositional Approaches

Software variability in compositional approaches comes from the selection of code units. Rather than being mixed together with common code as in annotative approaches, variable code is separated into code units. A code unit represents a feature of a configurable system and can be configured by the corresponding configuration option. A unit can be a line of code, a method, and a modular of implementing a specified functionality. These units of variable code are combined together with common code under different composition mechanisms [58, 30] such as extension and adapter patterns. Based on composition mechanisms, code units can be activated or inactivated by setting corresponding configuration options.

Analogously, we employ the *bicycle* example above to illustrate how compositional approaches work to make code configurable. In the example (see Figure 2.3), common code is class Bicycle. Configurable or variable code is class MotorBicycle. They are separated from each other and exist in different modules. The common code and configurable code are combined by extension, i.e., class MotorBicycle extends class Bicycle. Configurable code can be activated by selecting feature MotorBicycle in configuration setting of a system. A specific system variant can be generated by a selected set of feature modules.

```
1.      class Bicycle{
2.
3.              #ifdef Motor
4.              Engine engine
5.              #endif
6.
7.              void drive()
8.              {
9.                      ...//driving a bicycle
10.             }
11.
12.             #ifdef Motor
13.             void startEngine()
14.             {
15.                     ...// starting a motor engine
16.             }
17.
18.             void stopEngine()
19.             {
20.                     ...// stoping a motor engine
21.             }
22.             #endif
23.     }
```

Figure 2.2: Implementation of a bicycle including a feature Motor

2.1.2 Configuration Setting

Configuration setting is a process of determining option values following documentation. There exist three configuration user interfaces: command-line, configuration files, and configuration GUI.

Command-line is a powerful way for users to interact programs. Command line parameters can be used to set option values and alter the behavior of a program without having to modify and recompile its source code. But it normally applies to small program with a well defined behavior. For larger and more complex applications, the command line parameters are not sufficient to manage a large amount of configurations and behaviors in a system. In addition, command line interface is difficult for beginners.

```
1.      class Bicycle{
2.                  void drive()
3.              {
4.                      ...//driving a bicycle
5.              }
6.      }
7.
8.      class MotorBicycle extends Bicycle{
9.
10.             void startEngine()
11.             {
12.                     ...// starting a motor engine
13.             }
14.
15.             void stopEngine()
16.             {
17.                     ...// stoping a motor engine
18.             }
19.
20.     }
```

Figure 2.3: Implementation of a bicycle including a feature Motor

Configuration files are widely used to control the behavior of application programs. With user guidance documentation, users are allowed to specify values of configuration options in a configuration file which they are interested in. Users also can add comments for an option such as optional values and descriptions. For a large-scale application, each component or module is allowed to have its own configuration file, which can be loaded in different orders at initialization time or runtime. They also can be independently accessed at runtime. This flexibility gives users more choices to customize behavior of applications. Nowadays, configuration files represent the primary means of configuration setting.

The configuration files have some limitations as well though they are the primary configuration means in today's software systems. Configuring via a file, users are assumed to know what and how to configure, e.g., predefined syntactic structures, valid option values and their formats. In other words, configuration files cannot provide guidance and feedback on specifying the value of an option. They can be silent when users introduce some errors, e.g., typos in a option name or value.

GUI is a user-friendly way for configuration setting. Configuration GUI often provides wizard to guide users to identify and configure desired options. For introduced errors from users such as typos, violating formats, and invalid values, GUI is able to give response in time and offer suggestion for correct configuration. This brings a lot of convenience to users. Many applications, e.g., Windows operating systems, Firefox browsers, and various kinds of web services, adopt configuration GUI by offering option menus or wizards. However, this convenience comes at a large cost. Maintaining options is very expensive as software evolves because option menus or wizards need to be changed or reset if some options are added or removed.

Configuration data normally exists in two ways: ad-hoc configuration files and central databases. In the ad-hoc representation, there is no single configuration data format. An application has its own configuration data format such as XML, INI, and JSON. Linux is a representative system of using ad-hoc representation for configuration data, where there exist hundreds of configuration-specific data formats.

Central databases are used to store configuration data. For instance, Windows operating systems organize configuration data into a hierarchical database of key-value pairs, called *registry*. Relational databases are proposed to store configuration data as well in [77]. Configuration databases normally are used in large-scale applications which consists many modulars and have considerable amount of configuration options. A single configuration service is easier manage a large amount of options. However, in the database model, additional information for administrators such as default values, schema, and comments are not available. Besides, the maintenance of configuration options is costly due to the central control when software evolves.

2.1.3 Mapping

Mapping Time

Configuration data needs to be mapped to variable points in code base by predefined mechanisms, altering the behavior of a software system. Technically, the mapping happens at three points in time.

Compile time. Compile is the process of creating an executable program from source code. In the first stage of compilation, preprocessing replaces certain pieces of text by other text or conditionally omit some text in the source code according to system macros. This stage allows users to control variability of a configurable software system by setting variable points. Via these variable points, configuration

```
static struct config_int ConfigureNamesInt[] =
{
 ...
   {
         {   "temp_buffers", PGC_USERSET, RESOURCES_MEM,
              gettext_noop("Sets the maximum number..."),
              NULL,
              GUC_UNIT_BLOCKS
          },
          &num_temp_buffers,
          1024, 100, INT_MAX / 2,
          check_temp_buffers, NULL, NULL
   },
   ...
}
```

Figure 2.4: A code snippet in PostgreSQL-9.5.4

is able to determine which portions of source code of a configurable software system are translated into assembly instructions and converted into an executable program.

Initialization time. Initialization is the preparation of a program at program start-up, where predefined variable points in the code are initialized according to system environment and user preferences. Configuration at initialization time determines certain aspects of how the system or program is to function. Typically, these configuration option values are stored in initialization files or passed to a system via command line parameters.

Run time. Many configurable software systems provide interfaces to alter their behavior at runtime. By changing values of variable points in the code during software execution, a configurable system can be adapted to a dynamically changing environment or new user preferences. Runtime options are often more expensive in terms of maintenance costs than compile time or initialization time options. Changing an option value at runtime might require modifying values of dependent options in different layers.

Mapping Mechanisms

The mechanisms of mapping configuration data to variable points in code base vary in different software projects. For compile time options, option names are often defined

```
...
if(arg.equals("input") && i < args.length)
{
        inputFile = args[i++];
}
else if(arg.equals("output") && i < args.length)
{
        outputFile = args[i++];
}
else if(arg.equals("output-type") && i < args.length)
{
        outputType = parseType(args[i++]);
}
...
```

Figure 2.5: A code snippet in Log4J-configuration-converter

```
...
if (i % 2 == 0)
{
  reader = new SequenceFile.Reader(fs, path, conf);
}
else
{
  final FSDataInputStream in = fs.open(path);
  final long length = fs.getFileStatus(path).getLen();
  final int buffersize = conf.getInt("io.file.buffer.size", 4096);
  reader = new SequenceFile.Reader(in, buffersize, 0L, length, conf);
}
...
```

Figure 2.6: A code snippet in Hadoop 2.7.1

as macro variables in dedicated files. These macro variables are directly included in the source code. The preprocessing at compile time evaluates values of these macro variables to determine whether associated code is converted into an executable program. As shown in Figure 2.2, the macro variable *Motor* is an option. The variable can be directly defined in the source code or a dedicated file like *#define Motor*. If this option is set in the example, all code will be compiled.

For initialization and runtime options, developers often use clean interfaces to manage mapping information [87]. Typically there exist three interfaces of mapping configuration option values to program variables.

Structure-based. Structure-based mapping mechanisms often adopt data structures to directly bind configuration option names, option values, and the corresponding variables in the source code. The example shown in Figure 2.4 is used to illustrate how structure-based mechanisms work. The code snippet in the example is from PostgreSQL [6], an open source object-relational database system. As we see, the name of option *temp_buffers* and the corresponding variable *num_temp_buffers* are stored in the data structure *ConfigureNamesInt* and binded together. The program can obtain the value of option *temp_buffers* by accessing the variable *num_temp_buffers*.

Comparison-based. Instead of using data structures, comparison-based mechanisms identify an option by comparing its predefined option name in the program and option names in configuration data. If matched, the option value is mapped into the corresponding program variable. This mapping is typically used in command-line configuration mechanisms. The code snippet in Figure 2.5 shows how option values, under comparison-based mechanisms, are mapped to program variables in an open source project Log4J-configuration-converter [4]. If the option *input* is set in configuration data, the corresponding value is stored in variable *inputFile*.

Container-based. In container-based mechanisms, programs normally load all configuration data (all option names and values) and store it in a container like a hash table and provide a set of common *getter* methods which retrieve the corresponding option value by inputing the name of an option. By invoking such *getter* methods, option values can be mapped to the corresponding program variables. Container-based mechanisms are widely used in large software projects. For instance, Apache Hadoop [34] adopts this mechanism as well as its ecosystems. Figure 2.6 shows the code snippet from Hadoop 2.7.1. In the code snippet, the common *getter* method *conf.getInt* returns the value of option *io.file.buffer.size* and stores it into the variable *buffersize*.

Summary. This section presents configuration mechanisms in software systems, which consists of a configurable code base, configuration setting, and mapping. There mainly exist two kind of approaches for implementing configurable code base, annotative-based and compositional-based. Main manners of configuration setting are command-line, configuration files, and GUI. For configuration data mapping mechanisms, macro variables is the main manner at compile time. At initialization and run time, there are three kinds of approaches: structure-based, comparison-based, and container-based.

2.2 Static Program Analysis

```
1. int main()                          1. int main()
2. {                                   2. {
3.       int sum = 0;                  3.       int sum = 0;
4.       int i = 1;                    4.
5.       while (i<=100)                5.
6.       {                             6.
7.         sum = sum+i;                7.         sum = sum+i;
8.         i = i + 1;                  8.
9.       }                             9.
10.       printf("%d \n", sum);        10.       printf("%d \n", sum);
11.       printf("%d \n", i);          11.
12. }                                  12. }
```

(a) (b)

Figure 2.7: (a) An example program (b) A forward slice of the program the criterion (3, sum).

Static program analysis is the process of extracting facts about programs without the actually executing them. The application of static program analysis has spread into a variety of software engineering tasks such as software fault localization, testing and maintenance. In this thesis, we also use static analysis techniques to solve software configuration issues. Next, we introduce static analysis techniques applied in our work.

```
1. int main()                          1. int main()
2. {                                   2. {
3.       int sum = 0;                  3.
4.       int i = 1;                    4.       int i = 0;
5.       while (i<100)                 5.       while (i<100)
6.       {                             6.       {
7.         sum = sum+i;                7.         sum = sum+i;
8.         i = i + 1;                  8.         i = i + 1;
9.       }                             9.       }
10.       printf("%d \n", sum);        10.
11.       printf("%d \n", i);          11.       printf("%d \n", i);
12. }                                  12. }
```

(a) (b)

Figure 2.8: (a) An example program (b) A backward slice of the program the criterion (11, i).

2.2.1 Program Slicing

A program slice consists of the parts of a program which affect the values computed at some point of interest with respect to a slicing criterion. A slicing criterion often is a pair consisting of a line number and a variable in the line. The parts of a program which have a direct or indirect effect on the values computed at the slicing criterion are called the program slice of the slicing criterion. The process of computing program slices is called program slicing [75].

The conception of program slicing is first proposed by Weiser in 1979 [83, 82, 81, 15]. Weiser defines a slice as a reduced and executable program extracted from an original program by removing some statements, such that the extracted program can replicate part of the behavior of the original program.

The definition of a slice has been developed after decades. Nowadays a slice is defined as a subset of statements and control predicates of a program which contribute to the computed values at some point with respect to a slicing criterion, but which do not necessarily constitute an executable program. According to construction of a slice, there are two forms of slice: a *forward slice* and a *backward slice*. A forward slice contains the statements of a program which are potentially affected by the value of interest computed at a slicing criterion at runtime. A backward slice contains the statements of a program which can might have effect on the value of interest

computed at slicing criterion at runtime. The processes of computing a forward slice and backward slice are called *forward slicing* and *backward slicing*, respectively.

In order to understand the definition of a slice, we use an example to show which statements in a program are taken into a slice given a slicing criterion. Figure 2.7 (a) shows a program which calculates the sum of natural numbers from 1 to 100. Figure 2.7 (b) shows a forward slice of this program with respect to criterion (3, **sum**), where 3 represents the line number and **sum** indicates the variable of interest in the line. As we can see, all statements in the forward slice are affected by the value of variable **sum**. Whereas, all computations involving variable **i** are "sliced away".

Figure 2.8 shows a backward slice of the example program above. The criterion of the backward slicing is (11, **i**) and the backward slice is shown in Figure 2.8 (b). As we can see, all statements in the slice contribute to the value of variable **i** at the criterion.

Next we will introduce how to compute a slice given a specific criterion and involving conceptions.

Intraprocedural Slicing

Before presenting slice algorithms, we first introduce some basic conceptions used in the slicing.

Control Flow Graph (CFG). A CFG is a program representation which models all executions of a method by describing control structures. Specifically, a CFG of program P is a graph in which each node represents a statement from P and the edges indicate the flow of control in P. Each node n has two sets: $DEF(n)$ and $REF(n)$. $DEF(n)$ is the set of variables whose values are defined at the statement associated with node n. $REF(n)$ is the set of variables whose values are referenced at the statement associated with node n.

As an example, Figure 2.9 shows the CFG of the example program in Figure 2.7 (a). The graph shows all possible executions of the example program. Each node corresponds to a statement of the program, e.g. node 2 is associated with the statement at line 3 in Figure 2.7 (a). Node *Entry* corresponds to the beginning of the program. Node 4 represents a predicate, i.e., **while** statement at line 5, which controls the flow of execution. As we see, this node has two outgoing edges, one of which goes to node 5 and the other goes to node 7. Each node has two variable sets, e.g., $DEF(2) = \{sum\}$ and $REF(2) = \emptyset$ while $DEF(5) = \{sum\}$ and $REF\{i\}$.

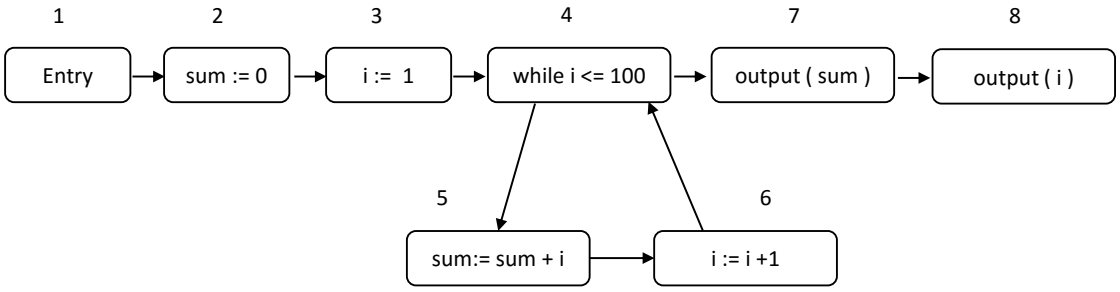

Figure 2.9: CFG of the example program of Figure 2.7 (a).

Data Flow Dependence. Data flow dependence (also called data dependence) is defined in terms of the CFG of a program. A node j is *data flow dependent* on node i if there exists a variable x such that:

- $x \in DEF(i)$,
- $x \in REF(j)$,
- there exists a path from i to j without intervening definition of x.

Back to the CFG of the example program in Figure 2.9, node 5 is data flow dependent on node 2 because (a) node 2 defines variable *sum*, (b) node 5 references variable *sum*, and (c) there exists a path 2 →3 →4 →5 without intervening definition of variable *sum*.

Control Dependence. Control dependence is defined in terms of the CFG of a program as well. Node j is control dependent on node i if:

- there exists a path from i to j such that any $u \neq i$, j in the path is *post-dominated* by j and
- i is not post-dominated by j.

where post-dominated is defined as following. A node m in the CFG is post-dominated by a node n if all paths from m to the end of the program pass through node n.

Typically the statements in the branches of an **if** or **while** are control dependent on the control predicate. In the CFG of the example program (see Figure 2.9), node 6 is control dependent on node 4 because there exists a path 4 →5 →6 such that: (a) node 5 is post-dominated by node 6, and (b) node 4 is not post-dominate by node

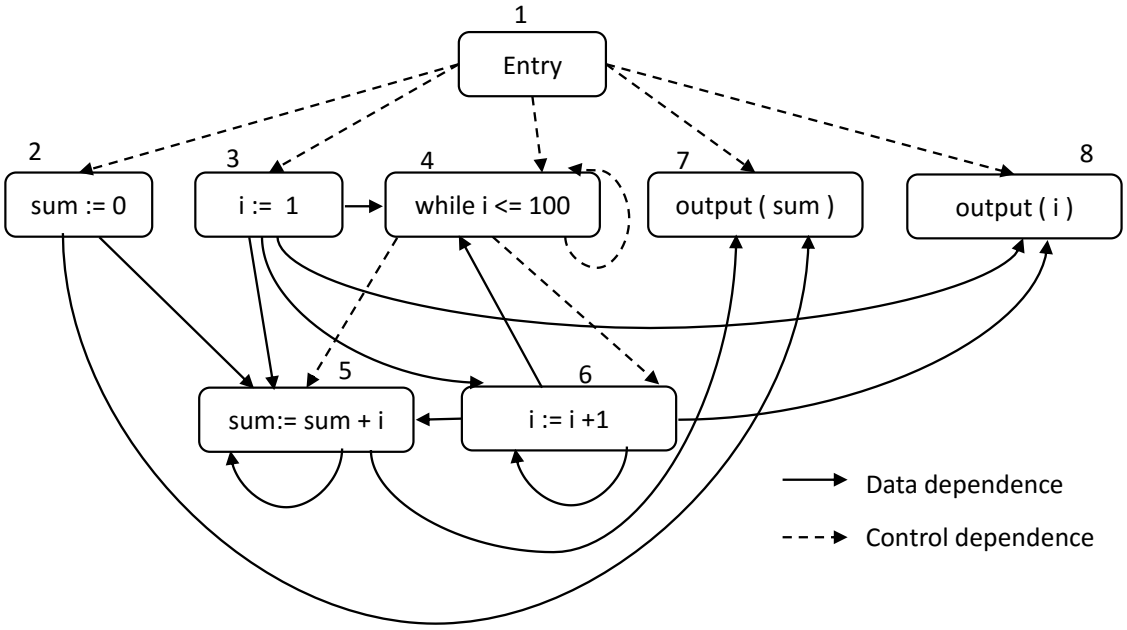

Figure 2.10: PDG of the example program of Figure 2.7 (a) .

6 since there is another path 4 →7 which goes to the end of the program without passing through node 6.

Program Dependence Graph (PDG). A PDG is an intermediate representation of making explicit both the data and control dependences for each statement in a program [29]. Given a single procedure program, the PDG of the program is represented as a graph whose nodes correspond to program statements and whose edges model dependences in the program. There is a directed edge between node v_1 and v_2 if the statement associated with v_2 is data flow dependent or control dependent on the statement corresponding to v_1 in the program.

Figure 2.10 shows the PDG of the example program in Figure 2.7 (a). Dashed arrows represent control dependence and solid arrows represent data dependence. All statements which are not in nested loop or conditional in the program are control dependent on *Entry* node. The statements in a nested loop or conditional are control dependent on the predicates of the loop or conditional. The statement associated with node 4 is control dependent on itself because the **while** loop could be executed more than one time.

Data dependences exist between statements of defining a variable and statements referencing the variable. The statement in node 8 is data dependent on the statement associated with node 6 because there exists an execution path from node 6 to node

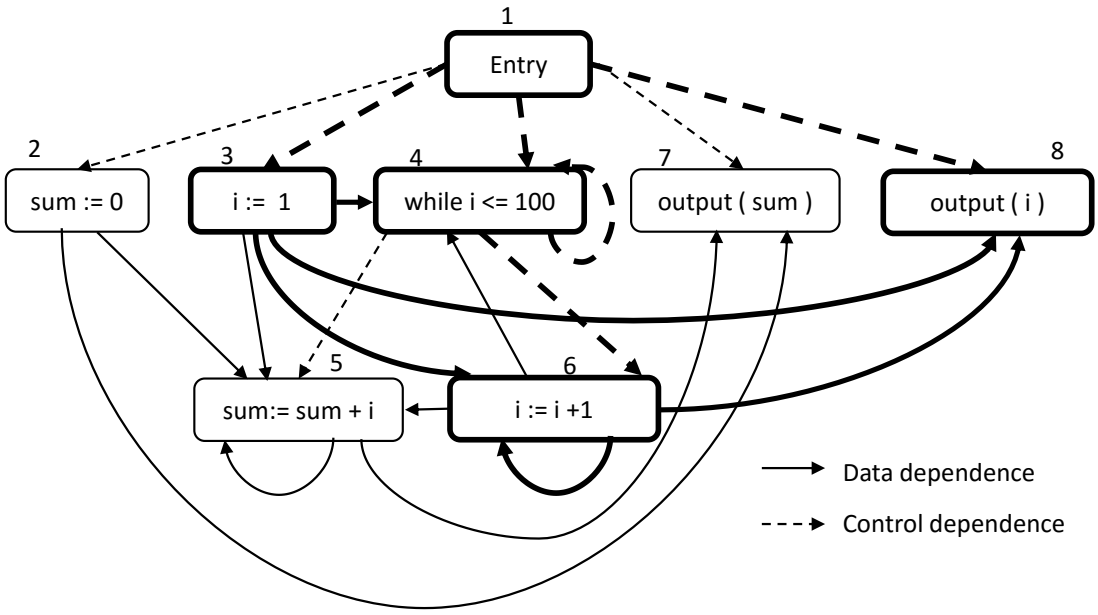

Figure 2.11: The backward slice of the example program in Figure 2.8 (a).

8 if the predicate in node 4 is satisfied. Meanwhile, the statement in node 8 is also dependent on the statement associated with node 3 if the predicate in node 4 is not satisfied. Consequently, the statement in node 8 is data dependent on the statements associated with node 3 and 6. The statements in node 5 and 6 are data dependent on themselves due to the multiple execution of the **while** loop.

Intraprocedural Slicing. Intraprocedural slicing is defined over the PDG of a program. The idea of using PDG to construct program slices was first proposed by Ottenstein and Ottenstein [55]. Horwitz et al. presented an approach of intraprocedural slicing based on a modified version of the program dependence graph of Ottenstein and Ottenstein [29, 8].

In dependence-graph-based approaches, the slicing criterion is converted to a node in the PDG from a pair of a line number and a variable in the line. For instance, the criterion (11, i) in backward slicing in Figure 2.8 (a) corresponds to the node 8 in the PDG shown in Figure 2.10.

Given a slicing criterion, denoted by node v of a PDG , the slice of the PDG with respect to v is a graph containing nodes on which v has a transitive data flow or control dependence. In other words, all nodes can reach v via data flow or control dependence edges.

Figure 2.11 shows the computation of the backward slice of the example program in Figure 2.8 (a). Nodes and edges in bold are the backward slice with respect to the criterion (8, i). Node 8 corresponds to the criterion in the PDG of the program. Nodes 3 and 6 directly reach node 8 via data dependence edges while node 1 directly reaches node 8 via the control dependence edge. Node 4 transitively reaches node 8 via control and data dependence edges, and node 6. Meanwhile, node 1 and 3 also transitively reach node 8 via other paths. All nodes which directly or transitively reach node 8 constitute the sub graph of the PDG, i.e., the backward slice with respect to criterion (8, i) in the program. Other nodes and edges not in bold are sliced away.

Interprocedural Slicing

Intraprocedural slicing does not consider facts that the value of interest at a criterion crosses the boundaries of procedure calls. It cannot be applied into multi-procedural programs. Addressing this issue, Horwitz, Reps, and Binkley proposed a new dependence graph representation of programs called *system dependence graph* which can be used to perform interprocdural slicing [37].

System Dependence Graph (SDG). SDG is an interprocdural extension of the program dependence graph representing multi-procedural programs. A multi-procedural program is considered as a single main procedure and a collection of auxiliary procedures. Parameters are passed by value-result among procedures. A SDG can be constructed by connecting PDGs of procedures in a multi-program based on parameters passing models.

In approach [37], parameter passing by value-result is modeled as follows. For a method call, the calling procedure first copies the values of actual parameters of the call site to temporary variables before the call. Then the called procedure initializes formal parameters by the corresponding temporary variables. In the called procedure, the final values of formal parameters are first copied to temporary variables before returning. The actual parameters in the calling procedure are updated by there temporary variables from the called procedure. Based on this model, PDGs of procedures in a multi-program are connected together and form the SDG of the program.

The SDG adds five different types of nodes to deal with a method call in the multi-procedural program. A call site in the procedure is represented by a *call node*. The operations which store the values of the actual parameters to the temporary variables in the calling procedure are indicated by *actual-in nodes*. Updating values

```
1. int main()                          13. int add(int x, int y)
2. {                                   14. {
3.       int sum = 0;                  15.     return x+y;
4.       int i = 1;                    16. }
5.       while (i<100)
6.       {
7.         add(sum, i);
8.         add(i, 1);
9.       }
10.       printf("%d \n", sum);
11.       printf("%d \n", i);
12. }
```

Figure 2.12: An example multi-procedural program.

of temporary variables from the called procedure to actual parameters in the calling procedure is represented by an *actual-out node*. The actual-in and actual-out nodes are control dependent on the corresponding call node.

In the called procedure, loading values of temporary variables from the calling procedure to formal parameters in the called procedure is indicated by a *formal-in node*. The operations which store returning values into temporary variable in the called procedure are presented by *formal-out nodes*. Formal-in and formal-out nodes are control dependent on the entry node of the called procedure.

In addition, the SDG adds interprocedural dependence edges due to parameter passing between procedures. A *parameter-in* edge is used to connect corresponding actual-in and formal-in nodes. A *parameter-out* edge exists between corresponding formal-out and actual-out nodes.

Figure 2.12 shows an interprocedural program which computes the sum of natural numbers from 1 to 100. Different from the example program shown in Figure 2.7 (a), the addition operator is implemented in another procedure *add*. The *main* procedure calls the add procedure to get the sum of two numbers.

Figure 2.13 shows the SDG of the example program above. The SDG consists of two parts, *calling procedure* and *called procedure*, which are connected by additional nodes and edges stated above. Each call in the main procedure corresponds a call node in the SDG. The call node is control dependent on the **while** node, at the same time, the entry node of the called procedure is control dependent on the call node. The passing of a parameter between procedures is represented multiple nodes. For

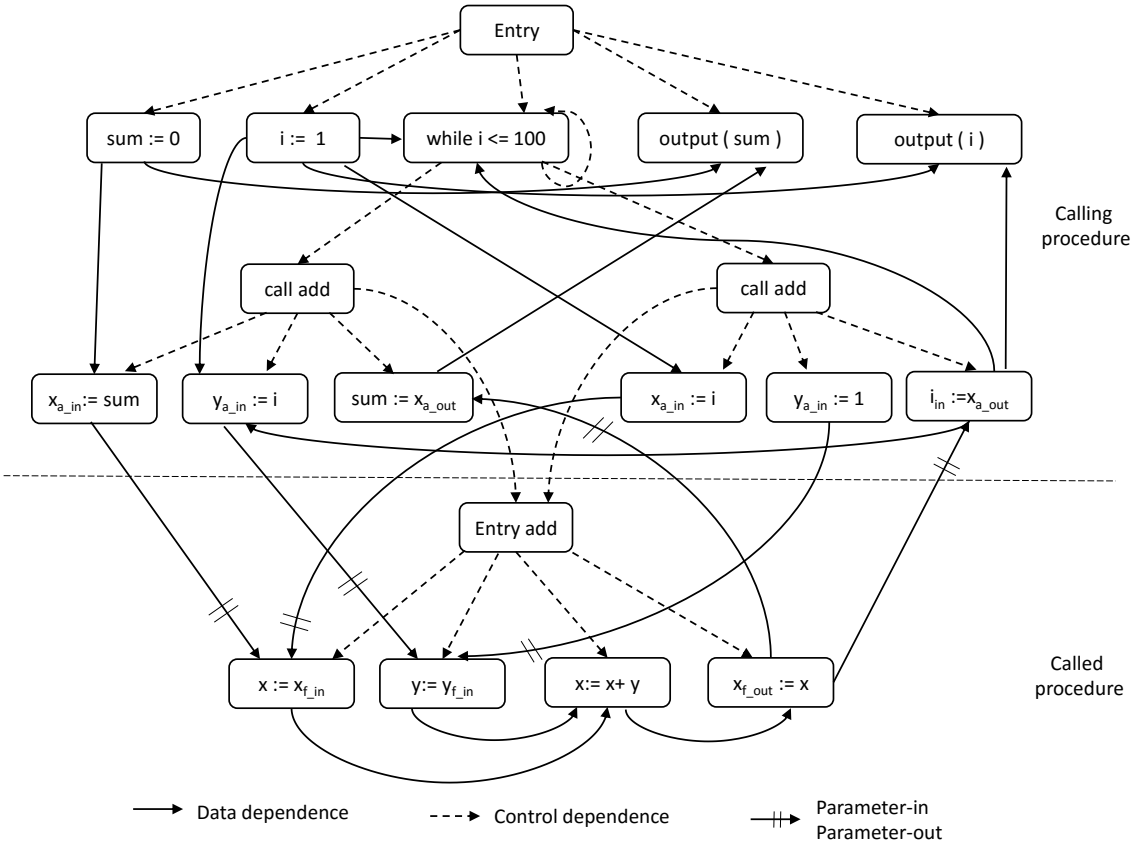

Figure 2.13: The SDG of the example prograom in Figure 2.12.

instance, the node x_{a_in} represents the operation of copying the value of variable *sum* to a temporary variable in the first procedure call. The x_{a_in} node is control dependent on the call node and data dependent on the node associated with initialization of the variable *sum*. In the called procedure, node x_{f_in} indicates loading the temporary variable to the corresponding formal variable x. There exists a parameter-in control dependence edge between nodes x_{a_in} and x_{f_in}. For the return of variable x, the node x_{f_out} indicates the copying operation from the formal variable x to a temporary variable in the called procedure. The node x_{a_out} presents the updating of actual variable *sum* by the returned value from the called procedure. The nodes x_{f_out} and x_{a_out} are connected by a parameter-out edge. Similarly, other parameters are handled in the same way.

Interprocedural Slicing. Interprocedural slicing is defined as a reachability problem using the SDG, just as intraprocedural slicing is defined over the PDG. The slices can be obtained by the improved Weiser's interprocedural-slicing method [81]. In the

improved approach, an additional process is needed before performing slicing, namely *computing summary edges*.

Summary edges. A summary edge is a transitive dependence edge, which connects an actual-in node with an actual-out node. The summary edge represents the dependence between the connected nodes which may be created in the called procedure or through other called procedures. A summary edge is added between an actual-in node and an actual-out if there exists a direct or transitive dependence between their corresponding formal-in and formal-out nodes in the called procedure. The algorithm of computing summary edges was presented by Reps et. al. [66].

After computing summary edges, the improved approach performs interprocedural slicing in two phases. In the first phase, the algorithm goes backwards along data dependence edges, call edges, summary edges, and parameter-in edges (no parameter-out edges) from the node associated with a slicing criterion. This phase the algorithm does not descend into the called procedures. In the second phase, the algorithm starts from all nodes reached in the first phase and goes backwards along data flow dependence edges, control edges, summary edges, and all parameter-out edges (no call edges and parameter-in edges). All nodes and edges visited in these two phases form the interprocedural slice of SDG with respect to a slicing criterion. The details of the algorithm was presented in [37]. Figure 2.14 shows the backward slice of the example program in Figure 2.12.

Forward intra-procedural and inter-procedural forward slicing is computed in the similar way. The slices are computed by tracing dependencies in the forward direction. We do not introduce detailed algorithms of forward slicing.

2.2.2 Thin Slicing

Large-scale object-oriented programs intensively use heap-allocated data and complex data structures. Slices of these programs include many internal implementation details of those data structures which are rarely useful to end programmers. The irrelevant statements make it difficult to trace the flow of data through the heap. To exclude the statements which are not useful for program testing. Manu Sridharan el at. [70] propose *thin slicing*.

A thin slice consists only of statements which contribute to compute and copy a value to a statement of interest. Statements that explain why these statements affect the statement of interest are excluded. For instance, with respect to a statement that

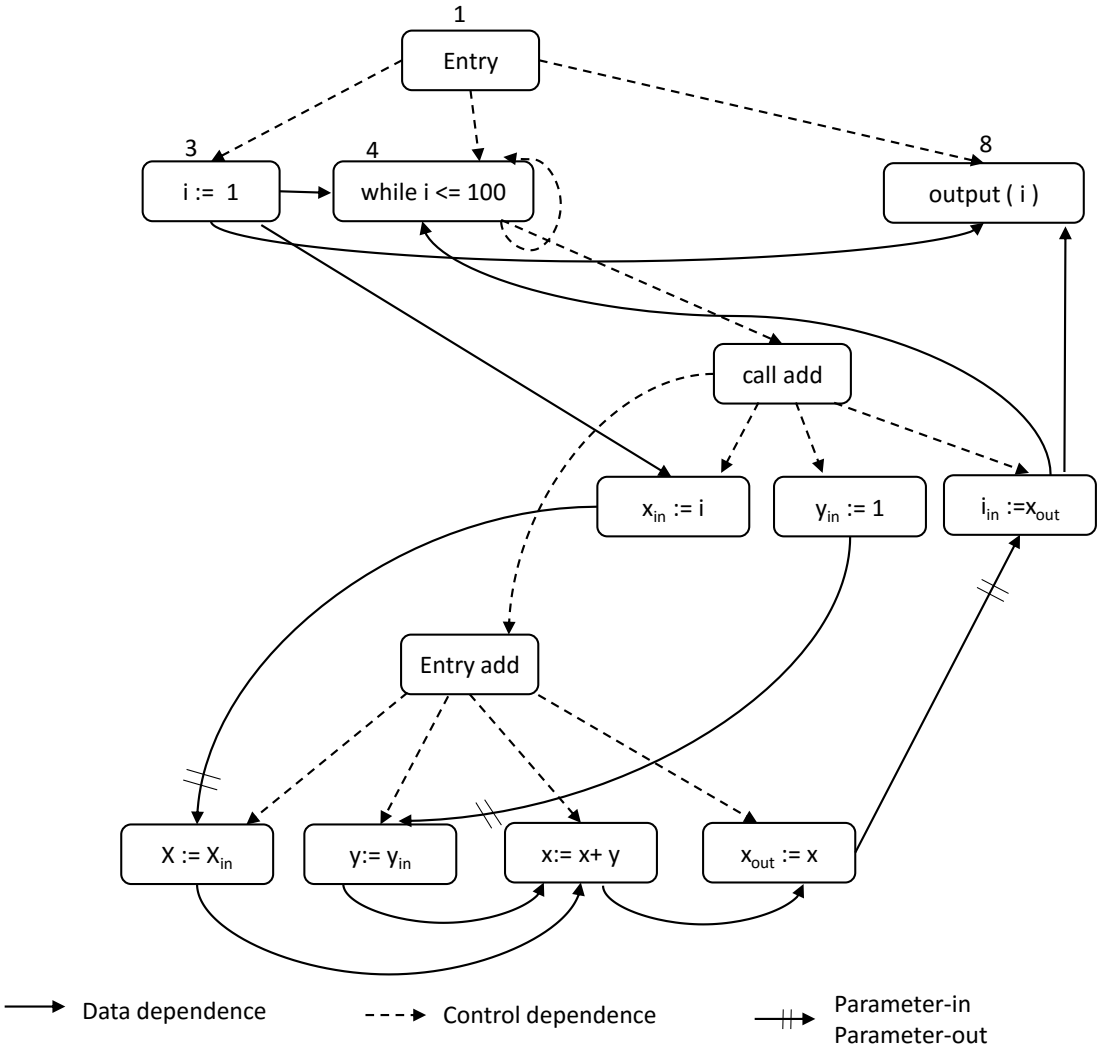

Figure 2.14: The backward slice of the example program in Figure 2.12.

```
1. x = new A();
2. z = x;
3. y = new B();
4. w = x;
5. w.f = y;
6 if (w==z){
7.  v=z.f; //  the statement of interest
8. }
```

Figure 2.15: An example program to illustrate thin slicing.

reads a value from a container object, a thin slice includes statements that store the value into the container, but excludes statements that manipulate pointers to the container.

In thin slicing, statements helping compute and copy a value of interest are called *producer statements* and non-producer statements in the traditional slice are called *explainer statements*. Producer statements are defined in terms of *direct uses* of memory locations, i.e., variables and object fields. A statement s directly uses a location l iff s uses l for some computation other a pointer dereference. For instance, the statement $y = x.f$ does not directly use x, but it does directly use $o.f$, where x points to o. A statement t is a producer statement for a statement s of interest iff (1) $s = t$ or (2) t writes a value to a location directly used by some other producer statements.

We use the example shown in Figure 2.15 to illustrate thin slicing. Line 7 is the statement of interest and directly uses field f of object z. Since w and z are aliased, line 5 writes a value into the field of interest. Line 5 is considered as a producer statement. Similarly, line 5 directly uses y, which is written at line 3, making line 3 a producer statement as well. Consequently, line 7, 5 and 3 are comprised of thin slice for a line 7. In contrast, the traditional slice for line 7 is the entire example program.

2.2.3 Call Graph

An important data structure in program analysis is *call graph*. The construction of a call graph often is a prerequisite of many inter-procedural analysis including program slicing, data flow analysis, and so on.

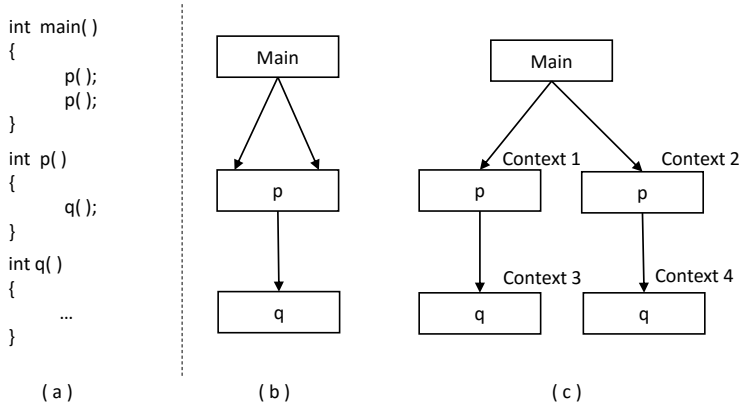

Figure 2.16: A example call chain of procedures and its context-insensitive and context-sensitive call graphs.

The call graph of a program is a directed graph that represents the calling relationships between the program's procedures. Each node in the call graph indicates a procedure and has an indexed set of call sites. Each call site is the source of zero or more edges to other nodes, representing possible called procedures of that site. A call site might points to multiple procedures for a dynamically dispatched message send or an invoked application[33, 29].

A call graph can be *context-insensitive* or *context-sensitive*. In a context-insensitive call graph, each procedure is represented by a single node. In a program, a procedure might be called at multiple places. All these call sites point to the node associated with the procedure. There is no distinction for these different call sties. In a context-sensitive call graph, a procedure is analyzed separately for different calling contexts. For each calling context, a different node represents the procedure. A procedure might have multiple context-sensitive versions which are represented by difference node in the call graph. The construction and analysis of context-sensitive call graphs are normally more expensive than that of context-insensitive call graphs, but analyses on context-sensitive call graphs are more accurate.

Figure 2.16 show a example call chain of procedures and its context-insensitive and context-sensitive call graphs. In the context-insensitive call graph, two call sites of procedure p in procedure *main* are treated in the same way and both are linked to node q. In the context-sensitive call graph, two call sites of procedure p are identified differently. Each call site corresponds to its own version of procedure p.

2.2.4 srcML-based Analysis

SrcML is an XML representation of source code for the efficient exploration, analysis, and manipulation of large software projects [20]. In the representation, source code is wrapped with information from AST (tags) [42] to form a single XML document. This representation provides full access to the source code at the lexical, structural, and syntactic levels. The facts about a program can be extracted by querying its the srcML representation. Figure 2.17 shows a **while** statement in Java and its srcML representation.

Summary. We introduce the definition of program slicing and related concepts including control flow graphs, data flow dependence, control dependence, program dependence graphs, and system dependence graphs. Program slicing can be intraprocedural and interprocedural. Intraprocedural slicing is based on program dependence graphs and interprocedural slicing is based on system dependence graphs. The slicing is defined as a reachability problem in a graph. We also use examples to illustrate how to compute a slice in a program graph and system graph. To improve the problem that a slice includes too many statements for human consumption, thin slicing is proposed to focus on the data flow of a value of interest. A thin slice consists only of producer statements without explainer statements. We briefly introduce analyses of call graph and srcML-based.

```
while ( i >= 0 )
{
    string1[i] = string2[i];                source code
    i--;
}

-----------------------------------------------------

<while>while
    <condition>(
        <expr>
            <name>i</name>
            <operator>&gt;=</operator>
            <literal type="number">0</literal>
        </expr> )
    </condition>
    <block>{

        <expr_stmt>
            <expr>
                <name>
                    <name>string1</name>
                    <index>[
                        <expr>
                            <name>i</name>
                        </expr>]
                    </index>
                </name>                         srcML format
                <operator>=</operator>
                <name>
                    <name>string2</name>
                    <index>[
                        <expr>
                            <name>i</name>
                        </expr>]
                    </index>
                </name>
            </expr>;
        </expr_stmt>
        <expr_stmt>
            <expr>
                <name>i</name>
                <operator>--</operator>
            </expr>;
        </expr_stmt>
    }</block>
</while>
```

Figure 2.17: A example code in Java and its srcML representation.

Chapter 3

Related Work

There have been a good deal of work on software configuration errors, in which practitioners and researchers attempt to solve software misconfiguration issues in various perspectives. Generally speaking, these works can be divided into two categories, namely *misconfiguration prevention* and *automated diagnosis of misconfiguration*. Research on misconfiguration prevention focuses on reducing possible factors of causing configuration errors before they occur in the real world. Research on automated diagnosis of misconfiguration attempts to identify the root cause of a configuration error occurred in practice without manual effort or less. Next, we introduce the two kinds of research works.

3.1 Misconfiguration Prevention

Many factors might cause configuration errors such as defects in the design and implementation of configuration mechanisms in a software system and users' lack of experience. By taking measurements on these factors, configuration errors associated with these factors can be avoided or occur less in practice. These works are organized by the factors they address.

3.1.1 Alerting on Mistakes in Configuration Setting

Study [88] shows that a majority of misconfigurations (70-86% out of investigated 546 real world cases) are due to mistakes in setting configuration parameters. Brown et al.

suggest typos are omnipresent and a major problem [17]. Hence, alerting users when a mistake is introduced can avoid a large fraction of real world misconfigurations.

ConfAlyzer [61], Encore [91], SPEX [87], and works of Nadi et al. [52] and Xiong et al. [85] fall into this category and attempt to extract constraints over options from source code or configuration data. When these constraints are violated in configuration setting, users will get an alert and correct introduced mistakes.

ConfAlyzer focuses on extracting configuration option types such as *String, Boolean, Numeric*. The approach obtains option types by identifying data types of variables storing option values in the source code. These types are bonded to corresponding options in configuration setting environment. Mistakes on an option type will trigger an alert.

SPEX and the work of Nadi et al. not only extract option types but also infer constraints among multiple options by tracking down data flow of option values. Constraints can be high-level semantic types like an IP address format, the value range of an option, and the control dependency among multiple configuration options. These constraints are obtained by analyzing predicates associated with an option value in the source code.

Xiong et al. define a range mode which is capable of specifying constraints over options including the correlation among multiple options. At the same time, they propose an algorithm to automatically generate these constraints.

Encore goes further and considers the correlations between options of a program and parameters of its execution environment aside from constraints on option types and among multiple options. The core idea of Encore is to adopt predefined rule templates to learn constraints on options from a given set of sample configuration data including information of the execution environment.

The constraints extracted by this kind of approaches can be also used to detect potential misconfiguration by applying them to the configuration data of existing systems. Giving an alert on violating option constraints in configuration setting is a more efficient way to reduce software configuration errors.

A major challenge for these approaches of extracting constraints on options is the low accuracy of analyzed results. Many constraints are application-specific without common and concrete patterns. Some logics for complicated string manipulations are really difficult to be inferred as well as high-level semantics constraints. In addition, scalability is a major challenge. Most of those approaches use program analysis techniques like data flow analysis to extract such constraints. The value of an option

might go through multiple third-party libraries. The analyses considering all involved libraries have scalability issues in program analysis techniques.

3.1.2 Detecting Inconsistencies Due to Option Changes

Misconfigurations can be also caused by mistakes of developers. During software evolving, configuration options are often changed, removed, and added. These changes result in modification of code associated with these options. Incomplete modification of relevant code would cause inconsistencies on these options implementations, which will lead to configuration errors in the system execution.

LOTRACK [47] works on this issue by creating a map between options and their affecting code fragments in the program. With the map, developers are able to locate which code fragments should be improved when an option is changed. LOTRACK uses static taint analysis to track down the value of a configuration option from the site the value is loaded to the point where the value may influence control flow decision. Eventually these code fragments identified by taint analysis are mapped to the corresponding options.

SCIC [14] directly detects inconsistencies on accessing option values in multiple layers of a program. In many software programs, a configuration option can be specified or accessed in different layers, e.g., by a user interface menu and by application code. Inconsistencies on option operations in multiple layers could lead to configuration errors. SCIC uses crossing-languages static analysis to automatically identify such inconsistencies in the program and avoid these kind of configuration errors after the software release.

These approaches focus on misconfiguration caused by changes on code of implementing options. Those misconfiguration are often tricky because the code associated options could partially function, e.g., work in one layer but not in other layers. One of challenges for this kind of approaches is that program analysis has to cross program languages because implementations of options in different layers often adopt different program languages.

3.1.3 Detecting Vulnerability in Handling Misconfigurations

Normally software systems have the functionality of handling configuration errors. Effective implementations of handling misconfiguration are able to detect configuration errors and pinpoint root causes in the error message. Poorly implementations cause

systems to misbehave in a mysterious way such as crashing and hanging. Proactive detection of configuration errors can expose bad implementations of handling misconfigurations in the system.

ConfErr [44] automatically generates and injects realistic errors in a system configuration file, assesses the target's resilience, and reports the output of a system failure due to configuration errors. The reports help developers identify and improve the poorly implementation of handling misconfiguration.

ConfDiagDetector [94] detects inadequate diagnostic messages for software configuration errors. Similarly, ConfDiagDetector injects configuration errors into the software under test, monitors and reports outcomes under injected configuration errors. Moreover, ConfDiagDetector employs nature language process techniques to assess whether diagnostic messages are adequate.

Proactive detection is an effective way to improve reaction of systems to configuration errors, i.e., pinpointing root causes in the error message. A major challenge for these approaches is to generate system tests to trigger injected configuration errors.

3.2 Misconfiguration Diagnosis

Instead of preventing software configuration errors, this branch of work attempts to diagnose the root causes of software misconfigurations, i.e., incorrect configuration options for system failures due to misconfiguration. Approaches of this type can be classified into three broad areas: program analysis, comparison-based, and replay-based.

3.2.1 Program Analysis Approaches

Program analysis approaches often employ program analysis techniques to extract facts of interest from a program which imply the linkage between one or more options and symptoms of a configuration error of the program. Based on types of program analysis techniques, these approaches can be grouped into two categories.

Static-analysis-based

Static analysis extracts facts from a program without running the program. These facts are used to infer which options likely lead to a configuration error. The representative works in this type of research are Sherlog [89] and ConfAnalyzer [60].

Sherlog analyzes the program of a problematic system by leveraging information provided at runtime logs to infer the execution path of the program prior to failure. The execution path might be incomplete but contains information which must have happened during failed execution. The must-have-happened information guides a developer to locate the root cause of a configuration error.

ConfAnalyzer uses data flow analysis to track down values of all options in a program and compute all statements in the program which are possibly affected by the value of each option. Then the affected statements are mapped to the corresponding options. With this map, a user is able to query the statements in the error message of a system failure and obtain options which might lead to the configuration error.

Approaches of this type do not require re-execution of a program but take as input the program and symptom information of a configuration error to infer options of possible incorrect setting. They are more acceptable in practice compared to dynamic analysis approaches because re-executing the program is not required. Re-executing the program requires specific input and environment information, which is expensive to be collected in practice. The major challenge of static analysis approaches is the high rate of false positives in diagnosis results. The false positives are mostly caused by the imprecise static analysis.

Dynamic-analysis-based

Addressing the low precision of static analysis, some approaches adopt dynamic analysis techniques which yield more precise results. They use dynamic analysis techniques to analyze the behavior of a system in question and infer the root causes of configuration errors. Two instances of prior work, ConfAid [12] and X-Ray [10], fall into this category.

ConfAid uses dynamic taint analysis to track down tokens from specified configuration option values through data and control flow dependencies as the program in question executes. Based on the tracked data, ConfAid pinpoints the configuration tokens most likely to have caused the exhibited problem. Similarly, X-Ray employs the dynamic information flow tracking technique to estimate the likelihood that a

specific part code is executed due to potential root causes. X-Ray focuses more on performance issues of software systems.

Compared to approaches of using static analysis, this type of approaches could obtain more precise diagnosis results because dynamic analysis provides accurate insight on the behavior of being analyzed systems. However, they require the reproduction of an error and the profile of a system behavior prior to failure. This can be costly in practice.

3.2.2 Comparison-based Approaches

Another family of approaches diagnose configuration errors across computers. They compare configuration data or behavior of a non-working system with that of a working system. Based on differences among them, options which likely lead to failures are identified.

Strider [50] and PeerPressure [79] diagnose configuration errors by comparing configuration data in a non-working system with that in a working system in other computers. Strider constructs the set of configuration differences and selects options out of them which are read during failing executions, and considers these options likely to lead to failures of the system. PeerPressure goes a further step. Instead of requiring configuration data of a working system, PeerPressure takes a set of sample machines and collects configuration data of a system on these machines. With statistics model, PeerPressure computes the relative frequency of various configuration settings. The unusual configuration settings which are read during the failing execution are identified as likely root causes of the system failures.

ConfDiagnoser [92], ConfSuggester [93] and the work of Attariyan et al. [11] detect configuration errors by comparing the behavior of a non-working system with that of working systems. ConfDiagnoser instruments code into the system and profiles the behavior relevant to configuration setting of the system. Meanwhile, ConfDiagnoser collects similar behavior data from a set of working systems. With the comparison among them, the abnormal behavior from a non-working system is identified and options associated with abnormal behavior are taken as likely root causes of the configuration errors. ConfSuggester uses similar approaches but focuses on configuration errors caused by software evolving. By comparing behavior between different versions systems, the options which are changed and relevant to the distinguished behavior in the later version are considered as likely root causes of misconfigurations. Attariyan et al. obtain the behavior of a system by running predicates for testing and com-

pare behaviors between the working system and the non-working system to diagnose configuration errors.

Comparing configuration data or behavior of a non-working system with that of working systems is an effective way to diagnose software configuration errors since it can significantly narrow down suspect configuration options. The challenge for this type of approaches is to collect a large amount of configuration data or behavior of working systems.

3.2.3 Replay-based Approaches

One branch of research on misconfiguration diagnosis is replay-based. Replay-based approaches automatically try possible configuration changes in a sandbox and observe whether one system recovers from a configuration error. In order to narrow down the space of configuration changes, those approaches often take some prior correct configuration data as a base for possible changes. Chronus [84], AutoBash [72], and Triage [76] are all systems of this type.

Chronus uses snapshot techniques to collect and store configuration data of a system at each checkpoint. When a configuration error occurs, Chronus employs binary search to locate which configuration change caused the system to stop working. AutoBash uses OS-level speculative execution to test the behavior of a system when various configuration changes are set. AutoBash can potentially fix configuration errors in the background by applying correct value changes of configuration options. If necessary, changes can be rolled back until problems are solved. Triage leverages checkpoint and re-execution techniques to repeatedly replay the moment of a failure and capture the failure environment. With rolling back events prior to the failure, Triage can pinpoint the root cause of th failure by analyzing the relevant events.

Replay-based approaches diagnose software errors at user's site, being able to obtain state data of a system at failure moment as well as environment data. Moreover, they are allowed to re-execute the system with various changes to fix these errors. The major challenge for using them in practice is that these approaches require changing execution environment of a system.

3.2.4 Knowledge-based Approaches

Another important branch of research on diagnosing configuration errors is knowledge-based. Knowledge-based approaches build a set of configuration knowl-

edge for a system by mining configuration data or events associated with configuration of the system. With these knowledge, they are able to pinpoint the root cause of a configuration error, even suggest a solution by inputing its symptom data.

Knowledge on Relevant Events. One typical event associated with configuration is the configuration data access. Rules mined from events accessing configuration data can be used to diagnose a configuration error. CODE [90] monitors configuration-accessing events of a system and mines invariant rules. With these rules, CODE can detect deviant program executions by sifting through a voluminous number of events. Similarly, Bauer et al. detect access-control misconfiguration by mining historical data of accessing related resource in the log. With mined rules, they can eliminate a large percentage of such misconfigurations interfering with legitimate accesses.

Knowledge on Previous Misconfiguration Cases. Many research works [57, 28, 63, 35, 49, 80] attempt to learn knowledge such as features of a specific type of misconfigurations, from historical misconfiguration cases. This type of approaches often use machine learning or other statistical models to obtain the association between symptoms and the root cause of a configuration error from historical cases data. With this knowledge, they can suggest one or more root cause by querying symptoms of a configuration error. ConfSeer [57] is a representative of this type of techniques. ConfSeer maintains a database of articles about technical misconfiguration solutions. With this database, ConfSeer combines interactive learning, information retrieval, and natural language processing techniques to suggest solutions by taking configuration files in question as input.

This type of techniques obtain knowledge about configuration from historical data. Collecting configuration data requires a substantial effort and is difficult concerning data privacy.

Chapter 4

ConfDoctor: Automated Diagnosis of Software Misconfiguration

This chapter addresses software configuration errors due to mistakes in option values. Existing approaches for this problem face several challenges like constructing data bases of configuration data and requiring statues of working systems, to be deployed in practice. We explore using static analysis to solve this problem without challenges faced by existing approaches.

The remainder of this chapter explains configuration errors we address, presents our approach of automated diagnosis of configuration errors, describes the empirical evaluation of the approach, and discuss its limitations.

4.1 Introduction

This section explains the motivation of this work and our core idea for troubleshooting software configuration. Then, we pinpoint challenges in implementing the idea and offer the solutions for those challenges.

4.1.1 Motivation

Software configuration errors are one of the major causes of today's system failures [32, 53, 54, 87, 59, 40, 18, 27]. For instance, recently misconfiguration-induced outages have been reported from major IT companies, including Microsoft, Amazon and Facebook [73, 74, 41]. Moreover, end-users also suffer from various configuration er-

rors of software. The investigation in [88] shows that around 27% of the issues in one company's costumer-support database are labeled as configuration-related.

Many groups from industry and academia have been working on the automated diagnosis of software configuration errors. As stated in Section 3.2, those existing approaches require crucial data or operations, which are difficult to achieve in practice. Comparison-based approaches like Strider [50] and PeerPressure [79] collect a large amount of configuration data from different working instances of a system and diagnose a configuration error by comparing their distinct configuration data. Similarly, ConfDiagnoser [92], ConfSuggester [93] and the work of Attariyan et al. [11] diagnose a configuration error by using the execution profile data of many working systems. Knowledge-based approaches [90, 57, 28, 63, 35, 49, 80] attempt to learn configuration rules from historical data and use these rules to diagnose configuration errors. For these approaches, it is challenging to build data bases such as configuration and execution profile data since configuration from users are sensitive in terms of data privacy. Replay-based approaches like Chronus [84], AutoBash [72], and Triage [76] base the prior working state of a system and try all possible configuration changes to locate the root cause of a configuration error or even correct it. These approaches require changing the execution environment of a system.

Addressing challenges in prior work, the construction of a configuration data base and changes of the execution environment, we attempt to develop an approach of diagnosing configuration errors which does not require user configuration data and execution environment information so that we can deploy it in practice.

4.1.2 Core Idea

Our idea is inspired by the way how developers typically debug erroneous configuration settings using stack traces of an error. As shown in Figure 4.1, the first step is usually to check the top frame of the stack traces. The program site referenced by this frame is investigated whether it is "close" to a statement reading the value of some configuration option. If this is the case, the next step is to analyze the dependency between the program site and the read point of the option. If the analysis on the top frame of the stack trace does not pinpoint any root cause candidate, it will be repeated for each of the subsequent stack frames.

For the example in Figure 4.1, the top frame of the stack traces is indexed by t. The corresponding program site is line 24. There is not any statement loading an option value after inspecting code around the line 24 in the method. The analysis

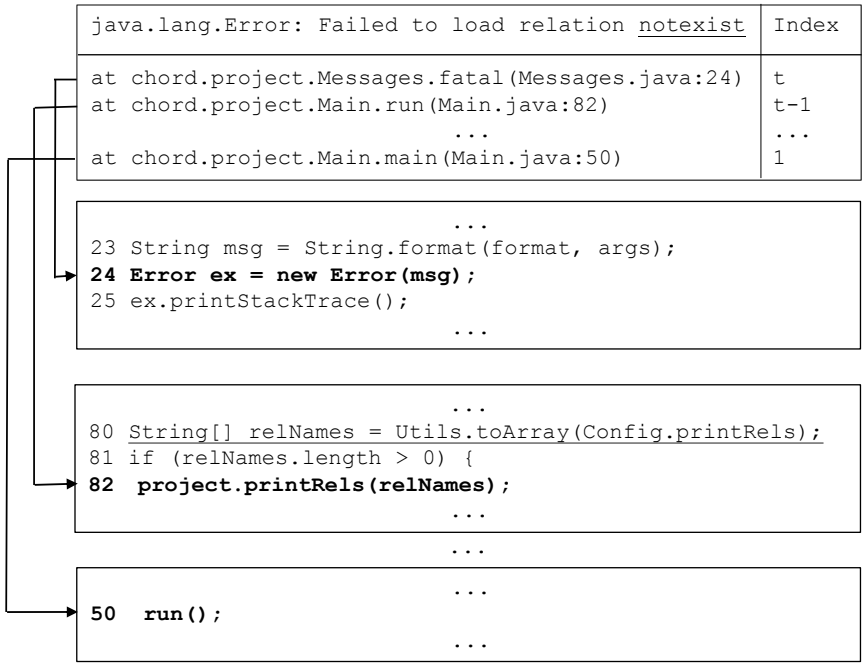

Figure 4.1: Example showing how developers diagnose a configuration error based on the stack trace. The statements in bold are program points referenced by the stack trace entries. The statement underlined is a read point of a configuration option.

goes to the second top frame $t-1$. The corresponding program site is line 82. The statement at line 80, near by the line 82, loads the value of option "Config.printRels". Further analysis shows the option value loaded at line 80 goes into the statement at the line 82. Consequently option "Config.printRels" is considered the potential root cause of the configuration error.

Given a configuration error, our idea is to identify whether there exists an possible execution path from where an option value is read to the point of an error raising by using static analysis. The options whose values go to the error raising site via a possible execution path are considered the root cause candidates of the configuration error. Then we analyze how likely these possible execution paths to the site of error raising are executed in the failing run and rank the corresponding options based on the possibility that their values go to the site of error raising. The ranked options are reported to users. The top options are more likely to be wrongly set and lead to the configuration error.

4.1.3 Challenges and Solutions

Our idea is to use static analysis to analyze the possible execution paths of an option value flowing to the site of error raising. There are two challenges for implementing our idea.

Scalability of Static Analysis

We target software systems which have a large amount of configuration options. These software systems usually are large scale. For instance, Apache Hadoop and HBase have over million of lines code and rely on dozens of libraries. Analyzing such systems raises scalability problems. Precise analysis on those systems cannot be completed because the amount of computation and memory required by analysis exponentially increases.

To deal with this challenge, we give up precise analysis and adopt coarse-grain analysis. We do not observe some behavior of software systems in static analysis. For data in a container like Array and List, we treat the container as a unit instead of distinguishing each element in the container. Such measurements can improve scalability of static analysis.

Precisely Identifying Possible Execution Paths

Static analysis observes the behavior of a software system, considering all possible executions. A phenomenon would happen that there exist many options whose values flow to the site of error raising in static analysis. This analysis leads to inaccurate diagnosis results, i.e., providing too many options to users as potential the root cause of a configuration error.

To solve this issue, we use *chopping* analysis. Specifically, we first use forward slicing to analyze all statements which might be affected by an option value. Meanwhile, we use backward slicing to identify all statements which have affected the statements referenced by stack traces of an error. Then we take the intersection of this two sets of statements to infer a possible execution path.

Moreover, we adopt a model to compute the correlation degree of each option with a configuration error. The core idea of the model is that an option is more likely to be the root cause if its value flow into the site of error raising through less methods. For instance, if the statement loading the value of option C_1 and the site of error

raising are in a method. The value of option C_2 goes through multiple methods via method calls and flows to the site of error raising. The option C_1 is more likely to be the root cause of the error than option C_2.

4.2 Problem Statement

Our work addresses one type of parameter-related misconfigurations, i.e., a crashing error caused by the incorrect value of a single configuration option. This type of configuration errors are a major part of misconfigurations from users according to a recent study [88].

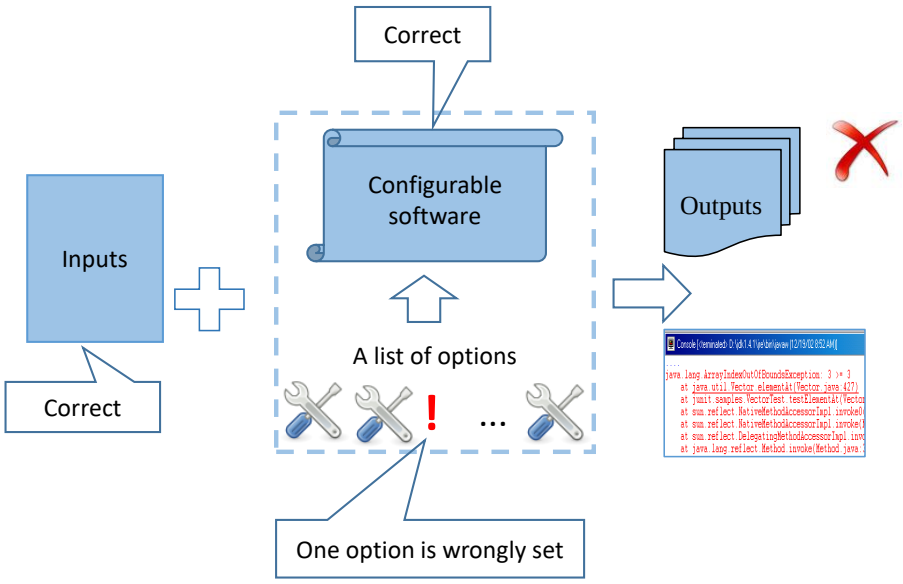

Figure 4.2: The scenario of the configuration error we address.

This type of configuration errors are described in Figure 4.2. Specifically, we work on the released software systems. The software systems are assumed to be well tested and rarely fail due to software bugs. Given that the input is correct, the system crashed with the incorrect results and the error message due to the wrong value of a configuration option. Our aim to identify which option is wrongly set from a large amount of options, e.g., hundreds or even thousands.

4.3 ConfDoctor Approach

Our approach considers the source code of a program as a set of *statements* S_1, S_2, ... Each statement is identified by a unique *program point*, also called a (program) *site*; thus, two `println`-statements at different sites are seen as different.

We consider configurations as a set of key-value pairs, where the keys are strings and the values have arbitrary type. This schema is supported by POSIX, Java Properties and Windows Registry, and is used in a range of projects [60].

For a program, we denote n configuration options of a debugged application by $c_1, \ldots, c_n$. For option c_i, we call a statement (program point) which reads-in value of c_i an *option read point* and denote it by ORP(c_i). Note that for each c_i they might exist multiple option read points.

In the following, non-capitalized letters (e.g. i, j, n, t) represent integers or configuration options (c_1, c_2, ...), letters P, R, S, Q denote statements, M, N are methods, and X, Y, Z are sets.

4.3.1 Overview

Our approach, called ConfDoctor, implies a following diagnosis workflow. For a targeted application we first perform a one-time configuration propagation analysis (Section 4.3.2) to identify statements possibly affected by the value of each configuration option. Given a crashing error and its error stack traces (Figure 4.1) we conduct a backward slicing analysis (Section 4.3.3) to identify statements which impact program points referenced by this trace. As next, an intersection of both sets of statements is computed (Section4.3.4). We use this result to correlate each configuration option with a given error with stack traces (Section 4.3.5). Finally, a list of configuration options ranked by the correlation degree is reported to users.

4.3.2 Configuration Propagation Analysis

This section describes how ConfDoctor analyzes the propagation of each option value in the program. ConfDoctor uses forward slicing techniques to track down the data flow of an option value and identifies all statements affected by the option value. The propagation analysis mainly consists of two steps.

Searching Option Read Points. We assume that configuration options of a software program are published. According to the configuration option list, our approach locates all option read points by searching configuration option names in the source code of the corresponding version.

Propagation Analysis. As stated in Section 2.2.1, program slicing allows to track down the data flow of a value of interest by identifying the data dependencies in the dependence graph. To identify all statements affected by a configuration option we use a static technique called *forward slicing* [82]. For a *seed* statement S, it attempts to identify the set of all statements (called *forward slice* $\mathrm{FS}(S)$) affected by the execution of S.

We deploy a variant of forward slicing which considers data dependence without control dependence (Section 4.4). The reason is that considering control dependence includes in slices $\mathrm{FS}(S)$ too many statements which are only indirectly affected by a configuration option. This might lead to a decreased accuracy of the diagnosis, which was confirmed by our evaluation.

For a particular configuration option c_i, the forward slicing analysis is conducted by using *all* option read points of the option c_i as seeds. Consequently, we define the *merged forward slice* $\mathrm{MFS}(c_i)$ as the union of all forward slices over all option read points of c_i:

$$\mathrm{MFS}(c_i) = \bigcup_{S \text{ is } \mathrm{ORP}(c_i)} \mathrm{FS}(S).$$

4.3.3 Stack Trace Analysis

In this section, we describe how to analyze parts of the program associated with a single execution using static analysis. The stack traces of an error record called methods in the execution before crashing. Using the stack traces, static analysis is able to identify the parts of being possibly executed in the program. Specifically, ConfDoctor adopts program slicing techniques to analyze all statements which have affected the statements referenced by the stack traces.

A typical stack trace is an ordered list of size t pointing to statements in nested methods called up to the point of failure. Each such referenced statement is called a *frame execution point* and is denoted by $\mathrm{FEP}(j)$, for $j = 1, \ldots, t$. We index stack trace entries from bottom to top, i.e. from the main method to the method where

an exception occurs (see Figure 4.1). Thus, FEP(t) is the program site where an exception has been raised, and FEP(1) is in the `main`-method.

To identify statements which have influenced program points referenced by a stack trace, we use *backward slicing* [82], a static analysis technique analogous to forward slicing. For a *seed* statement S, the *backward slice* BS(S) is a set of all statements whose execution might have influenced S.

Our stack trace analysis considers *all* frame execution points, not just the (top) FEP where the exception is raised. Consequently, we treat each FEP as a seed and compute its backward slice. The results are used to obtain a *merged backward slice* MBS which is a union of all backward slices:

$$\text{MBS} = \bigcup_{j=1}^{t} \text{BS}(\text{FEP}(j)).$$

Our stack trace analysis focuses on analysis in the application program and does not consider tools or third-party libraries. If a frame execution point does not reside in the source code of the application program, our technique is able to automatically exclude it using the package name.

Contrary to forward slicing, our implementation of backward slicing considers *both* data dependence and control dependence. The primary reason is that a stack trace records the execution path before an error occurs. It reflects the program's flow of execution. Without considering control dependence, the stack trace analysis would miss statements affecting FEPs.

4.3.4 Chopping Analysis

Configuration propagation analysis tracks down the data flow of each option value and identifies statements affected by the values of all options. Stack trace analysis identifies parts of possibly being executed in the program in a failure run of a system. We infer whether an option is correlated to a configuration error by checking if there is an overlap between statements affected by the option values and statements possibly be executed prior to the error.

The core idea of ConfDoctor is to identify configuration options c_i for which there exists an execution path between some ORP(c_i) and some FEP(j). As illustrated in Figure 4.3, if an intersection of forward slice of ORP(c_i) and a backward slice FEP(j) is not empty, such execution path might exist. Since we have multiple ORPs

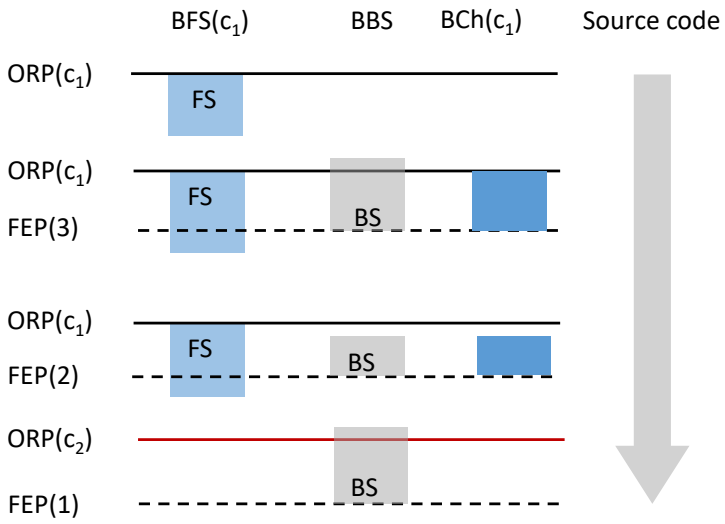

Figure 4.3: An example illustrates how the option read points ORPs of configuration options c_1 and c_2 and frame execution FEPs of an exception give rise to the merged forward slice MFS(c_1) of c_1, the merged backward slice MBS, and the merged chop MCh(c_1).

(per option) and multiple FEPs, the following definition is needed. For a given configuration option c_i the *merged chop* MCh(c_i) is the intersection of the merged forward slice MFS(c_i) and the merged backward slice MBS:

$$\text{MCh}(c_i) = \text{MFS}(c_i) \bigcap \text{MBS}.$$

4.3.5 Correlation Degrees

In the chop analysis above, there might exist multiple configuration options which are correlated to an error. Which option of them is more likely to be the root cause of the error? This section describes two variants of metrics used for ranking of configuration options based on the results of the chopping analysis. We first introduce some definitions.

Method Distance. As stated in Section 2.2.1, a static *call graph* CG of a program is a directed graph where each node represents a method and a directed edge (M, N) stands for method M calling method N. In CG, the *method distance* $d_{\text{meth}}(S_p, S_q)$ of two statements S_p and S_q is 1 plus the length of the shortest undirected path in

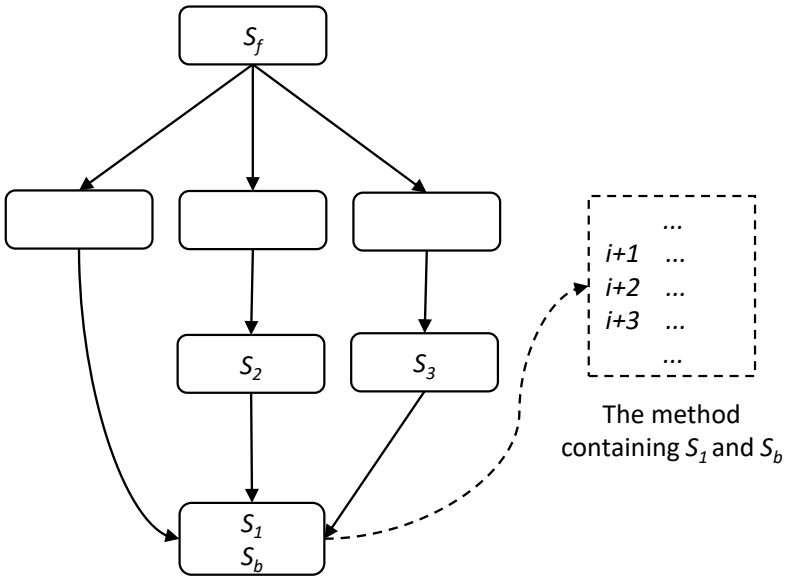

Figure 4.4: A fragment of a call graph with call paths from the method containing S_f to the method containing S_b

CG between a method containing S_p and a method containing S_q. $d_{\text{meth}}(S_p,\ S_q) = 1$ if both S_p and S_q are within the same method.

Method distance is used to estimate the "closeness" of any statement in a merged chop MCh(c_i) from an ORP or a FEP. We illustrate this in Figure 4.4 showing a partial call graph (each node represents a method). Let statement S_f be one of the ORP(c_i) and statement S_b one of the FEPs for a fixed configuration option c_i. In Figure 4.4 the top node labeled by S_f represents the method containing the statement S_f (analogously, statement S_b is in a method represented by the bottom node).

Furthermore, assume that S_1, S_2, S_3 are statements in the intersection $\text{FS}(S_f) \bigcap \text{BS}(S_b)$. Thus, $d_{\text{meth}}(S_f,\ S_1) = 3$, $d_{\text{meth}}(S_b,\ S_1) = 1$, $d_{\text{meth}}(S_f,\ S_2) = 4$, $d_{\text{meth}}(S_b,\ S_2) = 2$, $d_{\text{meth}}(S_f,\ S_3) = 3$ and $d_{\text{meth}}(S_b,\ S_3) = 2$. Obviously these statements differ by their distance to S_f and to S_b.

Forward Dependency Degree. Let c_i be a configuration option with a non-empty merged chop. Furthermore, let S_f be an option read point ORP(c_i) and S_b be a frame execution point FEP such that the forward slice of S_f has a non-empty intersection with the backward slice of S_b. We define a *forward dependency degree* $D_{fw}(S_f, S_b)$ as follows. Let S be a statement in $\text{FS}(S_f) \cap \text{BS}(S_b)$ with the smallest method distance $d_{\text{meth}}(S_b,\ S)$ to S_b, and in case of ambiguity with the smallest method distance

$d_{\text{meth}}(S_f,\ S)$ to S_f. Then $D_{fw}(S_f, S_b)$ is

$$D_{fw}(S_f, S_b) = (1/d_{\text{meth}}(S_f, S) + 1/d_{\text{meth}}(S_b, S)) * (1 + w)$$

where w is set to 1 if S and S_b are in the same source line (and so same method), and $w = 0$ otherwise.

In the example in Figure 4.4, S_1 is closest to S_b among statements S_1, S_2 and S_3. Further, S_1 and S_b are in same source line. Consequently, w is set to 1 and so $D_{fw}(S_f, S_b) = (1/3 + 1) * (1 + 1) = 8/3$.

Backward Dependency Degree. With the meaning of c_i, S_f and S_b as above, we define a *backward dependency degree* $D_{bw}(S_f, S_b)$ as follows. Let S be a statement in $\text{FS}(S_f) \cap \text{BS}(S_b)$ with a smallest method distance $d_{\text{meth}}(S_f,\ S)$ to S_f, and in case of ambiguity with a smallest method distance $d_{\text{meth}}(S_b,\ S)$ to S_b. Then $D_{bw}(S_f, S_b)$ is

$$D_{bw}(S_f, S_b) = (1/d_{\text{meth}}(S_f, S) + 1/d_{\text{meth}}(S_b, S)) * (1 + w)$$

where w is set to 1 if S and S_f are in the same source line, and $w = 0$ otherwise.

The intuition behind forward (and analogously backward) dependency degree is the following one. Value of $D_{fw}(S_f, S_b)$ is larger if the method distances of statements "affected by S_f" (i.e. in $\text{FS}(S_f)$) to S_b can be small. Furthermore, if S and S_b are in same source line (i.e. $w = 1$), then S_f can directly reach S_b, i.e., S_b is contained in the forward slice of S_f. All these cases indicate a higher probability that the particular option c_i (giving rise to S_f and S_b) can be responsible for the fault.

Simple Correlation Degree

We introduce a metric for ranking configuration options which is based on a sum of a largest forward dependency degree and a largest backward dependency degree.

For a fixed configuration option c_i let X be the set of all ORPs of c_i and Y be the set of all FEPs. For such a c_i the largest possible value $D_{fw}(S_f, S_b)$ of a forward dependency degree can be found by considering all combinations of S_f and S_b. This gives rise to a definition of a *forward correlation degree* $Cor_{fw}(c_i)$:

$$Cor_{fw}(c_i) = \max\left(D_{fw}(S_f, S_b) \mid S_f \in X,\ S_b \in Y\right).$$

We set $Cor_{fw}(c_i) = 0$ if there is no pair $S_f \in X$, $S_b \in Y$ with $\text{FS}(S_f) \cap \text{BS}(S_b) \neq \emptyset$ (and so no $D_{fw}(S_f, S_b)$ is defined).

Analogously, we define a *backward correlation degree* $Cor_{bw}(c_i)$ as

$$Cor_{bw}(c_i) = \max\left(D_{bw}(S_f, S_b) \mid S_f \in X,\, S_b \in Y\right).$$

Again we set $Cor_{bw}(c_i) = 0$ if there is no pair $S_f \in X$, $S_b \in Y$ with $\mathrm{FS}(S_f) \cap \mathrm{BS}(S_b) \neq \emptyset$ for c_i.

Our first metric for ranking configuration options called *simple correlation degree* Cor is the sum of the forward and backward correlation degrees:

$$Cor(c_i) = Cor_{fw}(c_i) + Cor_{bw}(c_i).$$

For evaluation purposes we also collect data about a pair S_f (a ORP) and S_b (a FEP) which maximizes $Cor_{fw}(c_i)$ or $Cor_{bw}(c_i)$. For a fixed c_i, let $(S_f^{fw}, S_b^{fw}) = \mathrm{argmax}_{S_f,S_b} D_{fw}(S_f, S_b)$ and let $(S_f^{bw}, S_b^{bw}) = \mathrm{argmax}_{S_f,S_b} D_{bw}(S_f, S_b)$. Then the *minimal ORP to FEP distance* $d_{min}(c_i)$ is a smaller one of the method distances $d_{\mathrm{meth}}(S_f^{fw}, S_b^{fw})$ and $d_{\mathrm{meth}}(S_f^{bw}, S_b^{bw})$. The index of the stack trace entry corresponding to S_b^{fw} (or to S_b^{bw} if $d_{\mathrm{meth}}(S_f^{bw}, S_b^{bw})$ is used) is called the *key frame* for c_i.

Correlation Degrees with Stack Order

The above-defined correlation degree does not consider the order of stack frames. However, the order of stack frames is significant for diagnosing the root cause of an error. A stack trace records the execution path in reverse "chronological" order from the most recent execution to the earliest execution. Paper [68] investigates 2,321 bugs from the eclipse project which are fixed in the method referenced by one of the stack frames. The result shows the number of bugs fixed in the *recently* executed methods is larger than that of ones fixed in the early executed methods.

$$f(j) = 1 - 1/j$$

To consider the impact of stack frame ordering we introduce an *stack order factor* f, where f is a weight function for each stack frame and j is the index of a stack frame (see Figure 4.1). The index j of the stack frame referencing the main method of a program is 1, yielding value $f(1) = 0$. We use the stack order factor to refine the definitions of the forward and backward correlation degree.

For the configuration option c_i the *forward correlation degree with stack order* is defined as:

$$Cor^{st}_{fw}(c_i) = \max\left(D_{fw}(S_f, S_b) * f(j) \mid S_f \in X,\, S_b \in Y\right)$$

where j is the index of the stack frame corresponding to S_b (while setting $Cor^{st}_{fw}(c_i) = 0$ if there is no pair S_f, S_b with $\mathrm{FS}(S_f) \cap \mathrm{BS}(S_b) \neq \emptyset$).

Analogously, the *backward correlation degree with stack order* is defined as:

$$Cor^{st}_{bw}(c_i) = \max\left(D_{bw}(S_f, S_b) * f(j) \mid S_f \in X,\, S_b \in Y\right)$$

where j is the index of the stack frame corresponding to S_b (with $Cor^{st}_{bw}(c_i) = 0$ if no $D_{bw}(S_f, S_b)$ is defined).

We are ready to define the main metric used for ranking of options, namely the *correlation degree with stack order*:

$$Cor^{st}(c_i) = \; Cor^{st}_{fw}(c_i) + \; Cor^{st}_{bw}(c_i) \; .$$

4.3.6 Ranking Configuration Options

As noted in Section 4.3.1, after the error stack trace is available, we compute the correlation degrees for all configuration options as described in Section 4.3.5. We can do this either using the simple correlation degree Cor or the correlation degree with stack order Cor^{st}. Since the latter choice produces better results, we consider it as the "final" metric of our approach (results for Cor are still shown in the evaluation).

A ranked list of configuration options obtained in this way is reported to users, with top entries (highest values of Cor^{st}) indicating the most likely root causes of a failure.

Summary. ConfDoctor mainly consists of two parts for diagnosing a configuration error: extracting statements from programs and computing correlation degrees of each option with the error. For the statement extraction, ConfDoctor identifies statements from programs which might be affected by an option value and statements which might have affected statements referenced by the stack traces of the error. The intersections of these two sets of statements are computed.

For the computation of correlation degrees, there are two models to computing correlation degrees of an option, Cor and Cor^{st}. Model $corst$ considers the order of stack traces of an error. ConfDoctor adopts model Cor^{st} since our evaluation shows diagnosis results are more accurate with Cor^{st} than with Cor. Finally, ConfDoctor outputs a list of options ranked by their correlation degrees with a configuration error. The options ranked top are more likely to be the root cause of the error.

4.4 Implementation

We implemented a Java-based prototype, called ConfDoctor, on top of the WALA library [78]. WALA is a static analyzer tool developed by IBM. WALA provides static analysis libraries for Java bytecode and related languages and for JavaScript including pointer analysis, call graph construction, inter-procedural data flow analysis, context-sensitive tabulation-based slicer, and so forth.

ConfDoctor uses the slicer in WALA to perform forward slicing and backward slicing analyses. In the forward slicing analysis, the heap dependence is ignored for improving the scalability of the slicer. Meanwhile, the control dependence is ignored as well because we focus the data flow analysis of option values in the configuration propagation analysis. For backward slicing, we consider the control dependence to use the order information provided by stack traces. Specifically, in the implementation, the DataDependence parameter is set as NO_BASE_NO_HEAP_NO_EXCEPTIONS for both slice types. For the ControlDependence parameter we use NONE for forward slicing, and FULL for backward slicing.

We also use WALA to compute the (static) call graph needed for the method distance d_{meth} computation. To achieve higher precision, we use here a more expensive algorithm, the Control Flow Analysis 0-CFA [33]. While our analysis focuses on the code in the application program, WALA needs to take into consideration Java libraries for building the call graph. Without considering e.g. callback methods, the call graph can be incomplete or imprecise.

Finally, we employ a database management system (specifically, MySQL) to store data about statements. Such data includes (among others) the fully-qualified class name, line number in the class file, and the method distance of each statement.

Table 4.1: Benchmark applications.

Program (version)	Lines of Code	#Options
JChord (2.1)	23.4 k	79
Randoop (1.3.2)	18.6 k	57
Hadoop (0.20.2)	103.6 k	141
HBase (0.92.2)	187.4 k	91

4.5 Evaluation

In this section we evaluate our approach and its implementation named ConfDoctor under the following aspects. First, we evaluate the effectiveness of the whole approach by investigating the rank of the actual root cause in the diagnosis results. Second, we evaluate the precision of both ranking metrics: the simple correlation degree Cor and the correlation degree with stack order Cor^{st}. Third, we explore the impact of static analysis with different types of dependence analyses on the accuracy of diagnosis results. Finally, we make a comparison with our previous work ConfDebugger [26].

4.5.1 Experimental Setup

Subject Applications

We evaluated ConfDoctor on four Java programs shown in Table 4.1. Column "Lines of Code" is the number of lines of code counted by CLOC [19]. For Hadoop and HBase, it is the number of lines of Java source code. The code in other program languages is not considered. Column "#Options" is the number of available configuration options.

JChord [39] is a program analysis platform for Java byte code initiated by Mayur Naik and Alex Aiken at Stanford University [60]. Randoop [64] is an automatic unit test generator for Java maintained by a product group at Microsoft. Apache Hadoop [34] comprises a distributed file system, a MapReduce implementation, and a job scheduling/cluster management framework. Apache Hbase [36] is a distributed, scalable, big data store which runs on top of Apache Hadoop's file system.

Configuration Errors

We collected 29 configuration errors [1] listed in Table 4.2. We evaluated all the configuration errors we found. The errors for JChord are taken from [60], and also used

[1]Their detailed description can be downloaded from the web site http://goo.gl/npOCVC.

Table 4.2: Configuration errors used in our evaluation.

Application	Id	Error description
JChord	1	No main class is specified
	2	No main method in the specified class
	3	Running a nonexistent analysis
	4	Invalid context-sensitive analysis time
	5	Printing nonexistent relations
	6	Disassembling nonexistent classes
	7	Invalid type of reflection
	8	Wrong classpath
Randoop	9	No testclass is specified
	10	Invalid type of output cases
	11	The value of alias-ration is out of bounds
	12	No method list is specified
	13	The tested method has missing arguments
	14	Incorrect name of the tested method
	15	Invalid symbols in name of output dir
	16	File name contains invalid symbols
Hadoop	17	Carriage return at the end of URL
	18	Old data dir after formatting namenode
	19	Wrong host name of master node
	20	Usage of *http* instead of *hdfs* in URL
	21	The storage dir of namenode not readable
	22	Missing the ¡property¿ tags
	23	Info port is in use by other process
	24	Missing port in the URL
HBase	25	Wrong port of the rootdir URL
	26	Wrong host name of the rootdir URL
	27	No permission of the data directory
	28	HMaster port is occupied
	29	Wrong port of ZooKeeper

in [92]. The data set contains 9 crashing errors. Since one of these errors is without a stack trace, we use the remaining 8 errors to evaluate our technique.

Randoop errors are injected by the tool ConfErr [44]. For a working configuration of Randoop, we use ConfErr to insert some typographic errors into the value of one of the configuration options. If the program crashes and produces a stack trace for the erroneous configuration, we use the error in our evaluation.

Hadoop errors are real world misconfigurations which are collected by us from the web and our own experiences of using Hadoop. Most of them can be found

on the website Stack Overflow [71]. Among these, three are tricky configuration errors. Error #18 occurs due to the incompatible namespaceID. After formatting the namenode, the directory specified by "dfs.data.dir" should be removed. Presence of this directory triggers the failure. Error #21 occurs if Hadoop has no permission to access the storage directory. Error #23 is caused by a port being used by another application.

All HBase errors are from the website Stack Overflow [71]. Some of the collected misconfigurations belong to the same type. For instance, there exist multiple errors caused by an incorrect host name. We use only one per type in this work.

4.5.2 Overall Accuracy

We measure the accuracy of ConfDoctor by the rank of the (unique) defective configuration option (i.e. option with an incorrect value, the root cause of a failure) in a ranked list of suspects. Rank 1 is the best possible result. We consider a configuration option c_i a suspect if its merged chop $\mathrm{MCh}(c_i)$ is not empty (or equivalently, by definitions in Section 4.3.5, if $Cor^{st}(c_i) > 0$).

The main results are shown in Table 4.3. The two columns under "Rank of the root cause" contain pairs R/S where R is the rank of the actual root cause in a ranked list of suspects of size S (highest rank is 1). Column Cor^{st} shows results obtained by the correlation degree with stack order (main output of ConfDoctor). Column Cor shows the ranking by the simple correlation degree. If the value of the correlation degree is the same for a defective option and for some other options, we report the worst ranking for ConfDoctor and mark this by "*". "N" indicates that the list of suspects does not include the defective option. For computation of averages, each "N" is treated as half of the number of available configuration options. Columns under "Statistics for rank" show the minimal ORP to FEP distance $d_{min}(c_i)$ and the key frame (Section4.3.5) for the configuration option c_i ranked as first. In the column "key frame" we use notation K/F to indicate that the key frame value is K and the total length of the error stack trace is F. A "-" indicates that d_{min} and key frame are not defined.

Column Cor^{st} in Table 4.3 shows the final output of ConfDoctor. Overall, ConfDoctor is highly effective in diagnosing misconfigurations. It successfully pinpoints the root cause for 27 out of 29 errors. For 20 errors, the defective option has rank 1. For other 7 errors, root causes are ranked in the top four places.

Table 4.3: Experimental results. The two columns under "Rank of the root cause" show the rank of the actual root cause for each error by the two proposed metrics. Columns under "Statistics for rank" show the minimal method distance and the key frame in diagnosing an error.

	Id	Rank of the Root Cause		Statistics for Rank	
		Cor^{st}	Cor	d_{min}	Key frame
JChord	1	2/47	2/47	1	18/19
	2	1 /53	2/53	1	18/19
	3	1 /45	1/45	1	3/6
	4	1/57	2/57	1	6/8
	5	1/42	1/42	1	2/4
	6	1/37	1/37	1	3/4
	7	1/48	2/48	1	3/4
	8	* 22/47	30/47	3	16/19
	Average	3.8/47	5.1/47	1.3	8.6/10.4
Radoop	9	1/37	1/37	1	4/6
	10	1/35	13/35	1	4/4
	11	1/47	2/47	1	7/8
	12	1/39	1/39	1	3/5
	13	1/41	1/41	1	3/11
	14	1/41	1/41	1	3/13
	15	4/43	2/43	1	4/6
	16	1/38	1/38	1	3/5
	Average	1.4/40.1	2.8/40.1	1.0	3.9/7.2
Hadoop	17	1/7	1/7	1	6/8
	18	1/11	1/11	1	3/8
	19	2/7	2/7	2	4/9
	20	1/18	2/18	2	6/9
	21	2/16	2/16	2	6/8
	22	1/11	1/11	1	5/7
	23	3/6	3/6	2	4/9
	24	1/11	1/11	1	5/7
	Average	1.5/10.9	1.6/10.9	1.5	4.9/8.1
HBase	25	1/17	1/17	1	4/4
	26	1/17	1/17	1	4/4
	27	3/20	8/20	8	9/9
	28	N	N	-	-
	29	3/5	3/5	1	3/5
	Average	10.8/30	11.8/30	2.2	4/4.4

In the case of JChord, ConfDoctor succeeds to pinpoint the root cause with high accuracy for 7 errors. The rank of the root cause for error #8 is 22. Code inspection shows that the configuration option is used to set up the command line for a child process. The value of the configuration option directly flows into a system process. Our static analysis cannot capture the dependency between command line arguments and configuration options. Consequently, our tool is not able to discover a connection between the defective option and the exception and fails in diagnosing error #8.

For Randoop, ConfDoctor ranks the defective option as the first for all cases except for error #15. For error #15 (ranked 4th), our investigation reveals that two of the three configuration options ranked higher than #15 are related to the defective option: they determine the subpath of a path described by option #15. Consequently, we conclude that ConfDoctor pinpoints the root cause of this error effectively.

The average ranking of Hadoop is 1.5. Among the 8 errors, 5 are ranked first. The rankings for errors #19, #21 and #23 are 2, 2 and 3, respectively.

The accuracy of our tool is slightly worse for HBase. ConfDoctor diagnoses root causes of 4 out of 5 errors. Two defective options are ranked as first, and other two receive rank 3. For error #28, the defective option is not in the list of suspects. A manual analysis shows that error #28 is caused by an incorrect port of HMaster (a master server for HBase). After the option value is read it is immediately forwarded to Java library class without any processing. Since ConfDoctor does not analyze the JDK library, the root cause of this failure is not included in the list of suspects.

Summary. The evaluation shows ConfDoctor is effective in diagnose configuration errors. ConfDoctor can successfully diagnose 27 out of 29 errors from JChord, Randoop, Hadoop, and HBase. For 20 errors, the failure-inducing configuration option is ranked first.

4.5.3 Comparison of Accuracy of *Cor* and Cor^{st}

In this section we contrast and analyse the accuracy of the simple correlation degree Cor against the correlation degree with stack order Cor^{st}.

Effectiveness of Cor. Recall that this metric is solely based on the method distance involving option read points ORP and frame execution points FEP. Contrary to Cor^{st}, it does not consider the order of FEPs. Note that the ranking is based the sum of the forward and backward correlation degrees (Section 4.3.5).

The diagnosis results are shown in column *Cor* of Table 4.3. For most of the errors, the ranking of the root cause is in the top three of diagnosis results. The average ranks of the root cause are 5.1, 2.8 and 1.6 for JChord, Randoop and Hadoop respectively. The average rank of the root cause for HBase is very high (11.8) since error #28 could not be pinpointed.

As the metric is based only on the method distance, it is informative to consider the minimal ORP to FEP distance d_{min} defined in Section 4.3.5. As shown in Table 4.3, for all errors of JChord and Randdop the value of d_{min} is 1 except for error #8. As explained in Section 4.5.2, ConfDoctor cannot successfully diagnose error #8. For Hadoop and HBase, the value of d_{min} is larger, especially for error #27.

The primary reason is that an error stack trace contains only partial information and does not contain the data on complete past execution traces. Hadoop and HBase are complex distributed programs. An option value might be passed along many methods from being read to being used. The method reading the incorrect value of the option might not appear in the stack traces. For instance,the value of the option "fs.default.name" is passed through 5 methods from being read to being checked (see Figure 4.5). The method *getDefaultUri* takes responsibility for reading the configuration option. The get methods with different arguments perform intermediate processing of it. After multiple intermediate processing, it is checked in the method *createFileSystem*. This creates a situation that methods specified by the stack trace are far away from the method for reading configuration options.

Effectiveness of Cor^{st} . ConfDoctor uses Cor^{st} to produce its final ranking. This metric assigns a higher "importance" to FEPs pointing to methods executed more recently before a failure (or equivalently, to FEPs with smaller method distance to the actual program site which rised an exception).

Data in column Cor^{st} in Table 4.3 indicates that the considering this factor improves precision of diagnosis results, but the improvement varies among the applications. The average rank of the root cause improves from 5.1 to 3.8 for JChord, from 2.8 to 1.4 for Randoop, from 1.6 to 1.5 for Hadoop, and from 11.8 to 10.8 for HBase. To understand these differences, we analysed the usage and programming patterns in regard to the configuration options in all four applications.

JChord and Randoop adopt a mechanism of centrally initializing configuration options. Especially for Randoop, the *randoop.main.GenTests.handle* method contains 45 ORPs. Most configuration options are initialized in this method when the program starts. The method appears in stack traces of all 8 errors. In this case, the most

```
public static URI getDefaultUri(Configuration conf) {
return URI.create(
fixName(conf.get(FS_DEFAULT_NAME_KEY,"file:///")));
}
...
public static FileSystem get(Configuration conf)
throws IOException {
return get(getDefaultUri(conf), conf);
}
...
public static FileSystem get(URI uri, Configuration conf)
throws IOException {
...
return CACHE. get(uri, conf);
}
...
synchronized FileSystem get(URI uri, Configuration conf)
throws IOException{
...
fs = createFileSystem(uri, conf);
...
}
...
private static FileSystem createFileSystem(URI uri,
Configuration conf ) throws IOException {
Class<?> clazz = conf.getClass("fs." +
uri.getScheme() + ".impl", null);
...
}
```

Figure 4.5: Code excerpt from Hadoop: the value of the configuration option "fs.default.name" is passed through 5 methods until it is checked

recent operation indicates the configuration option last processed. Consequently, considering the order of stack traces in Cor^{st} significantly improves the precision of diagnosis results.

Hadoop and HBase initialize a configuration option only when the module associated with the configuration option is loaded. Initializations of configuration options are scattered in the program and not concentrated in one or few methods. When a misconfiguration occurs, it does not involve many configuration options. For all errors, there are few configuration options which have the same or higher correlation degree with the root cause before applying the model. We conclude that the order of the stack trace does not improve the precision of the diagnosis results a lot.

The statistic key frame defined in Section 4.3.5 is also shown in Table 4.3. Notation K/F indicates that the key frame value is K and the total length of the error stack trace is F. Data shows that for the top ranked configuration option the key frame is within the "top" half of the error stack trace, i.e. closer to the method where an exception is thrown than to the "main" method.

Summary. With model Cor^{st}, ConfDoctor achieves more accurate diagnosis results than using model Cor. For JChord and Randoop, the average rank of the root cause is improved from 5.1 to 3.8 and from 2.8 to 1.4, respectively. For Hadoop and HBase, the average rank of the root cause is improved from 1.6 to 1.5 and from 11.8 to 10.8, respectively. With model Cor^{st}, the precision improvement on the diagnosis results varies on applications. For applications which centrally use option values, Cor^{st} can significantly improve the precision of diagnosis results but improve less for applications which sparsely use option values in the program.

4.5.4 Impact of Variants of the Dependence Analysis on Accuracy

ConfDoctor mainly relies on static analysis technique to diagnose the root cause of a configuration error. The accuracy of diagnosis results depends on whether the program slicing technique can precisely identify statements relevant for error propagation. The type of dependence analyses has a significant impact on the accuracy of diagnosis results.

In this section we evaluate the implementation choices stated in Section 4.4 by comparing the precision of ConfDoctor under different types of dependence analyses. We use a tuple (F, B) to indicate a type of dependence analyses. If the forward

Table 4.4: The diagnosis results with different variants of dependency and ConfDebugger's diagnosis results. Pairs R/S indicate the ranks of root causes in diagnosis, where R is the rank of the actual root cause in a ranked list of suspects of size S (highest rank is 1).

	Id	Variants of Dependency Analysis				Conf-
		ConfDoctor	(Ctr, Ctr)	(Ctr, NCtr)	(NCtr, NCtr)	Debugger
J Chord	1	2/47	4/64	2/57	2/8	2/2
	2	1/53	1/66	1/57	1/8	2/2
	3	1/45	5/65	9/19	2/6	1/1
	4	1/57	1/69	1/26	1/11	1/1
	5	1/42	1/65	2/19	2/6	1/1
	6	1/37	2/65	2/8	1/1	1/1
	7	1/48	3/69	4/6	4/6	1/1
	8	22/47	28/64	28/57	N	N
	Average	3.8/47	5.6/65.9	6.1/31.1	6.6/15.6	6.1/13.2
Randoop	9	1/37	3/53	2/11	1/3	2/2
	10	1/35	1/54	1/8	N	N
	11	1/47	3/55	1/20	1/5	2/3
	12	1/39	4/54	1/7	1/3	1/1
	13	1/41	4/54	1/12	1/3	1/1
	14	1/41	4/54	1/13	1/5	1/1
	15	4/43	17/53	4/18	4/7	2/4
	16	1/38	1/54	1/2	1/1	1/1
	Average	1.4/40.1	4.6/53.9	1.5/11.4	4.9 /10.8	4.9/8.8
Haddoop	17	1/7	1/32	1/32	1/2	1/1
	18	1/11	1/29	1/3	1/1	1/1
	19	2/7	9/30	9/21	2/3	N
	20	1/18	1/36	1/31	1/1	N
	21	2/16	2/38	2/2	2/2	N
	22	1/11	1/34	1/29	1/5	1/1
	23	3/6	16/31	16/20	2/5	N
	24	1/11	1/34	1/29	1/5	1/1
	Average	1.5/10.9	4.0/33	4.0/20.9	1.4/3	36.1/36.1
HBase	25	1/17	1/33	1/1	1/1	1/17
	26	1/17	1/33	1/1	1/1	1/17
	27	3/20	3/33	N	N	16/20
	28	N	15/32	N	N	N
	29	3/5	3/32	N	N	3/5
	Average	10.8/30	4.6/32.6	28/55	28/55	13.4/30

```
475. if ((!output_tests.equals("all"))
                &&  (!output_tests.equals("pass"))
                &&  (!output_tests.equals("fail")))
   {
476.   StringBuilder = new StringBuilder();
477.   localStringBuilder.append("Option output-tests
                        must be one of ");
478.   localStringBuilder.append("all");
479.   localStringBuilder.append(", ");
480.   localStringBuilder.append("pass");
481.   localStringBuilder.append(", or ");
482.   localStringBuilder.append("fail");
483.   localStringBuilder.append(".");
484.   throw new RuntimeException(localStringBuilder.toString());
485. }
```

Figure 4.6: Excerpt of the Randoop related to error #10

slicing considers control dependence, F is *Ctr*, and *NCtr* otherwise. Analogously, if the backward slicing considers control dependence, B is *Ctr*, and *NCtr* otherwise. In this notation, the default type of dependence analyses used in ConfDoctor is written as (*NCtr*, *Ctr*). Similarly, other three dependence analyses are indicated as (*Ctr*, *Ctr*), (*Ctr*, *NCtr*) and (*NCtr*, *NCtr*).

The diagnosis results under other dependence analyses are shown in Table 4.4. Column "Variants of dependency analysis" shows the ranks of root causes obtained by Cor^{st} produced by variants of the dependence analysis types. It uses analogous notation as columns for Cor^{st} and for Cor. Finally, column "ConfDebugger" reports results for our previous work.

ConfDoctor achieves the most accurate results for JChord (3.8) and Randoop (1.4) when using the default dependence type (*NCtr*, *Ctr*). The accuracy of using the dependence type (*NCtr*, *Ctr*) is similar to that of using the dependence type (*NCtr*, *NCtr*) for Hadoop. But ConfDoctor totally fails to diagnose errors #10, #27, and #29 when using the dependence type (*NCtr*, *NCtr*), though it achieves a little more accurate results than using (*NCtr*, *Ctr*) for Hadoop. For Hbase, using the dependence type (*Ctr*, *Ctr*) obtains the most accurate results because it can diagnose the root cause of error #28. But the root cause of error #28 ranks very low (15/32), which is not useful in the real world.

The explanation is that forward slicing considering control dependence introduces too many statements which are only indirectly affected by a configuration option. On the other hand, for backward slicing, ignoring control dependence can miss the execution information contained by a stack trace. The example in Figure 4.6 illustrates this. Line 484 is the program point referred by stack trace frame closest to the exception of error #10. Line 475 is the initialization statement of the configuration option output_tests. Obviously lines 476 to 484 have control dependencies with line 475. If we use line 484 to perform a backward slicing without considering control dependencies, none of lines in the excerpt is contained in the slice except for line 484 because they have data dependencies with line 484. This analysis misses line 475 which directly affects line 484. The accuracy of diagnosis result significantly decrease. instead, ConfDoctor is able to pinpoint the root cause of error #10 by considering control dependencies.

Summary. The types of program analysis for extracting statements from programs have significant impact on the precision of diagnosis results. The comparison indicates ConfDoctor achieves more accurate results when using the dependence type (*NCtr*, *Ctr*). There are two major reasons. First, forward slicing without considering control dependence excludes less-relevant statements with an option value. Second, backward slicing with considering control dependence uses control flow information contained in stack traces.

4.5.5 Comparison with Our Previous Work

ConfDebugger [26] is our preliminary work to diagnose software misconfigurations by using static analysis. If one FEP of a stack trace is contained in the forward slice of an ORP of a configuration option, or an ORP of the configuration option is included in the backward slice of an FEP of the stack trace, ConfDebugger considers the configuration option as the root cause candidate. Contrary to ConfDebugger, ConfDoctor applies a systematic approach presented in Section 4.3.5 to compute the dependency between one configuration option and an error.

As shown in Table 4.4 (Column *ConfDebugger*), ConfDebugger achieves a similar accuracy for JChord. But for Randoop, Hadoop, and HBase it fails in many cases. The reason is that ConfDebugger does not consider incompleteness of a stack trace. For many cases, the statements reached by the ORP of a configuration option do not appear in stack traces (see Section 4.3.5) . In Hadoop and HBase, the depth of method calls is relatively large. A stack trace misses some executed program points

when an error occurs. Consequently, ConfDebugger has a very low success rate for these cases.

Summary. ConfDoctor achieves more accurate diagnosis results than our previous work ConfDebugger. The primary reason is that stack trace of an error could be incomplete. The statements affected by an option value might not appear in stack traces but exist in methods which are close to stack trace methods in the call graph. ConfDoctor considers these statements for computing correlation degrees of an option. ConfDebugger does not consider them.

4.5.6 Time Overhead of Diagnosis

Our experiments were conducted on a laptop with Intel i7-2760QM CPU (2.40GHz) and 8 GB physical memory, running Windows 7.

The time cost is shown in Table 4.5. Column "FS&Importing" shows time of forward slicing and importing into the database (in seconds; one-time effort per program). Column "BS&Importing" states time of forward slicing and importing into the database (in seconds). Column "Analysis" gives time of final analysis.

The forward slicing does not consider control dependence and is relatively fast. The maximum time is 357 seconds for Hadoop. The forward slicing is a one-time effort per program. The computed slices can be used for the diagnoses of other errors.

The backward slicing considers control dependence and needs more time. For an error, time for the backward slicing varies on the size of the stack trace. The maximum time is 978 seconds for error #27. The time of computing correlation degree is just several seconds. The total of diagnosing an error is less than 20 minutes.

Summary. Considering the cost of manually diagnosing configuration errors, ConfDoctor takes less than 20 minutes to diagnose an error. Given diagnosing a configuration error requires much longer in practice, we believe that our time overhead is acceptable.

4.5.7 Discussion

Limitations

Our technique has several limitations. First, we focus on a subset of configuration errors, where the incorrect setting of an option causes a program to fail in a determin-

Table 4.5: The time overhead of diagnosing a misconfiguration. Column "FS & Improting" indicates the time of forward slicing and importing statements into database for an application. Column "BS & Importing" represents the time of backward slicing and importing statements into database for each error. Column "Analysis" indicates the analysis time for each error. The time unit is the second.

Apps	Id	FS&Importing	BS&Importing	Analysis
JChord	1	102	692	2
	2	102	605	2
	3	102	150	<1
	4	102	258	<1
	5	102	33	<1
	6	102	30	<1
	7	102	60	<1
	8	102	628	2
Randoop	9	169	412	<1
	10	169	241	<1
	11	169	565	1
	12	169	821	<1
	13	169	752	2
	14	169	957	2
	15	169	431	<1
	16	169	202	<1
Hadoop	17	357	65	<1
	18	357	76	<1
	19	357	59	<1
	20	357	641	1
	21	357	90	<1
	22	357	32	<1
	23	357	115	<1
	24	357	35	<1
HBase	25	87	453	2
	26	87	467	2
	27	87	978	3
	28	87	114	<1
	29	87	119	<1

istic way and produce a stack trace. Second, the accuracy of our technique depends on the availability of a stack trace. The lack of stack traces decreases the accuracy. Third, our technique just provides suspects and cannot tell a user why and how the configuration option is incorrect. Besides, our approach cannot distinguish configuration errors from bugs in the source code. ConfDoctor still produces a ranking list for failures caused by a bug in the source code, which can be misleading. Fourth, some misconfigurations are caused by the incorrect setting of a combination of multiple configuration options, our approach cannot pinpoint how many configuration options lead to a error. Finally, our implementation and experiments are restricted to Java and cannot cross components written by different languages.

Threats to validity

There are two threats to validity in our evaluation. First, configuration errors for JChord and Randoop are created by using ConfErr [44] as typographic mistakes inserted into the value of a configuration option. Real configuration errors collected from websites cover several misconfiguration types such as numerical parameters and system paths. Such errors might be not representative. Second, the number of subject application programs is small, though the programs we used in the evaluation are created by developers from different organizations and institutions. Thus, we cannot affirm that the results can be generalized to an arbitrary program.

4.6 Summary

This chapter presents a practical technique, ConfDoctor, to diagnose software configuration errors. It first analyzes the program in question to characterize statements affected by reading the values of configuration options. In the case of a failure, the stack trace is investigated in order to find a potential link between the option read points and the program sites listed in the stack trace. The suspicious configuration options are reported after being ranked according to the "strength" of such links. ConfDoctor does not require users to reproduce errors. The only data needed from a failed execution is the error stack trace. This facilitates deploying our approach as a third-party service and is an essential advantage compared to existing approaches which require to reproduce errors or to provide testing oracles.

We collect 29 configuration errors on 4 applications (JChord, Randoop, Hadoop, and HBase) to evaluate ConfDoctor in several aspects. First, we evaluate the effective-

ness of our approach in diagnosing configuration errors. The results show ConfDoctor is highly effective in troubleshooting configuration. It successfully diagnoses the root causes of 27 errors, for 20 of which the root cause is ranked first in the output.

Second, we evaluate two proposed models for computing correlation degrees with an error, Cor and Cor^{st}. The model Cor^{st} considers the order information of stack traces of an error and considers options more likely to be root causes which might affect statements near to the crashing site at run time. Results show that ConfDoctor with model Cor^{st} achieves more accurate results.

Third, we evaluate the impact of different variants of dependence analysis on the accuracy of diagnosis results. Results show the combination of forward slicing with considering control dependence and backward slicing without considering control dependence obtains the most accurate results.

Fourth, we compare ConfDoctor with our previous work, ConfDebugger. Results show ConfDoctor achieve much more accurate results than ConfDebugger does. The primary reason is that ConfDoctor considers all possible relevant statements with an option value and adopts a model to systematically compute the correlation degree of an option with a configuration error. ConfDebugger only considers whether an option value affects statements referenced by stack traces instead.

Finally, we evaluate ConfDoctor on the time overhead of diagnosing a configuration error. For 29 errors, the diagnosis time is less than 20 minutes, which is reasonable for diagnosing a configuration error.

Chapter 5

ORPLocator: Locating Option Read Points

This chapter addresses software misconfigurations due to inconsistencies between configuration documents and the corresponding version of a software system. These inconsistencies usually occur as software evolves. Changes like the removing some options are not updated to documents in time. The setting of such removed options cannot create effects on the system as documents state. This kind of misconfiguration leads to the frustrated experience for a user.

We present an approach, called *ORPLocator*, for detection of these inconsistencies between source code and documentation based on static analysis. Our approach automatically identifies source code locations where options are read, and for each such location retrieves the name of the option. Inconsistencies are then detected by comparing the results against the option names listed in documentation.

The remainder of this chapter first briefly introduces our work, then presents details of ORPLocator and its evaluation, and finally discusses its limitations.

5.1 Introduction

This section first introduces the motivation of our work using a real world configuration error. Then we present our idea to detect inconsistencies between documentation and source code. Next, we pinpoint a challenge in implementing the idea and our solution for the challenge.

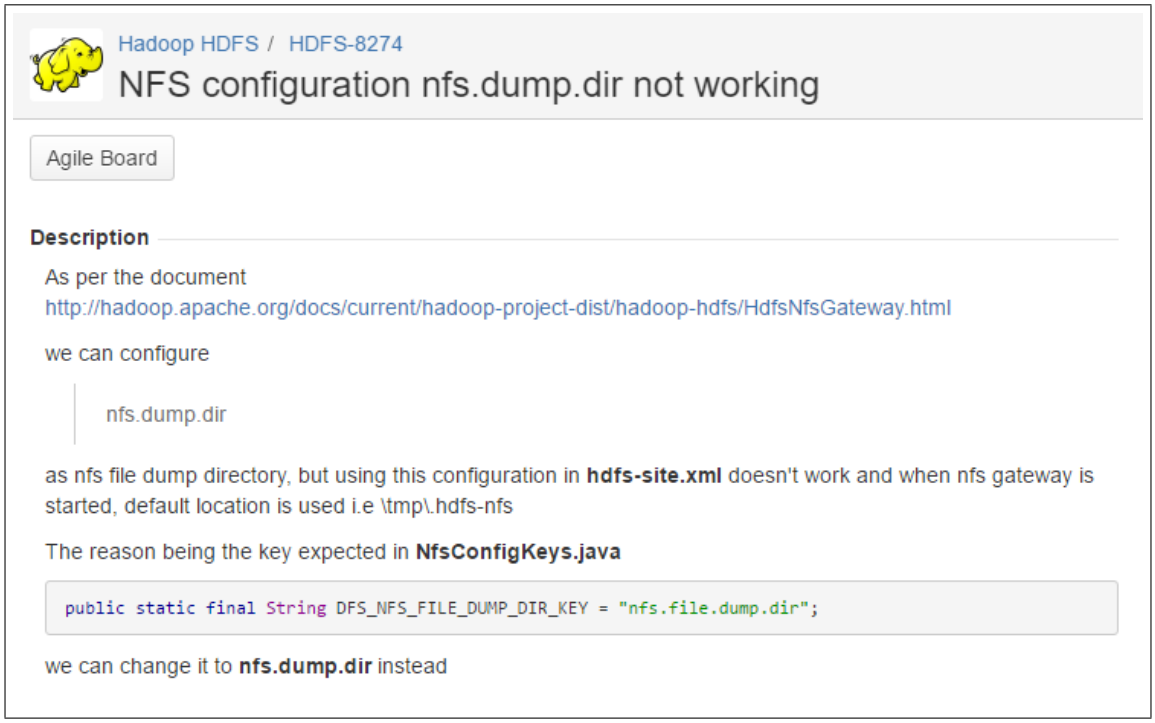

Figure 5.1: HDFS-8274: A real configuration issue in Apache Hadoop Distributed File System 2.7.0.

5.1.1 Motivation

Highly configurable software systems often provide documents for guiding users to configure them. With the documents, users are able to customize an application by setting some configuration options. One of the most frustrating things in the configuration setting is that the setting suggested by a document does not work, i.e., the setting of an option does not create the effect as the document states. A common reason for this is the specified option value is never read by the program.

Let us see a real world bug in Apache Hadoop Distributed File System (HDFS) 2.7.0 shown in Figure 5.1. The configuration document of HDFS 2.7.0 shows option "nsf.dump.dir" allows to set the dump directory of a Network File System (NFS) client for saving temporary data. However, after setting the option, HDSF still uses the directory under the default path instead of the path specified by this option. The root cause of the bug is that the option name is changed to "nsf.**file**.dump.dir" in the source code. The value specified by users is not read by HDFS.

Configuration issues of this type are common because the update of software documentation is slower than software evolution as stated in Section 1.2.2. The changed options in the source code are not updated to documents in time. These inconsisten-

CommonConfigurationKeysPublic.java

```
        ...
249: public static final String A =
250: "hadoop.security.service.user.name.key";
        ...
```

GroupMappingServiceProvider.java

```
        ...
34: public static final String B =
        CommonConfigurationKeysPublic.A;
        ...
```

CompositeGroupMapping.java

```
         ...
47:  public static final String C = B + ".providers";
48:  public static final String D = C + ".combined";
         ...
     public synchronized void setConf(Configuration conf) {
         ...
116:     this.combined = conf.getBoolean(D,true);
         ...
        }
         ...
```

Figure 5.2: A real case of how an option is used in Hadoop 2.7.1. Variable names are replaced by capitalized letters to improve readability.

cies between documents and source code lead to that a user configures configuration options of not being used in the software system. This type of configuration issues can frustrate users.

A straightforward approach for detecting inconsistencies between documented (i.e., "official") options and the corresponding version of source code is to check whether an option is used in the source code. An indication of this fact is that at least one code statement accesses (reads) the option value in the source code. We call such a code statement an *option read point* (*ORP*). To simplify, we do not consider whether the statement is dead code. Given a list of (automatically detected) ORPs together with corresponding option names, we can compare it against the documentation in order to detect the above-mentioned inconsistencies.

A common way for developers to locate option read points is to search the calls of the methods for reading options and infer option names directly from the string parameters used in a method call. This approach is not sufficient for complex applications. In the application we studied, the calls of option-reading methods usually take a variable as a parameter. Using existing approaches, detecting the name of the option in such cases requires to investigate many methods or classes.

A real example is shown in Figure 5.2. Line 116 in the class file *CompositeGroupsMapping.java* is an option read point in Apache Hadoop, which takes the class variable D as a parameter. The variable D is initialized by an expression of combining a string constant and another class variable C. Similarly, variable C is initialized by an expression of combining variable B and a string constant. However, variable B is declared in the super class *GroupMappingServiceProvider* of class *CompositeGroupsMapping*. Again, variable B is initialized by the variable A in a dedicated class for storing option names or their prefixes. In this dedicated class, variable A is initialized via a string. Finally, the value of variable D, the option name, is obtained as "hadoop.security.service.user.name.key.providers.combined". This analysis spans 3 classes and 2 packages. Such coding patterns are quite common in the applications which we studied.

Manually identifying option read points of a large amount of configuration options, e.g. thousands, takes a huge amount of human effort and is error-prone. Addressing this issue, we attempt to develop an approach for automatically identifying option read options for highly-configurable and large-scale software systems.

5.1.2 Idea

Given the name of a class C for handling (reading) configuration options and the source code of a program, we first mark the methods which fetch option values in the class. Then we identify all instances of the class in the program. All call sites of marked methods on these instances are located, i.e., all statements of reading option values are located. Finally, we infer which option each of these statements reads and output a map of option names and their read points. With this map, we identify inconsistencies between documents and source code by comparing identified options with documented options.

We use the example in Figure 5.3 to illustrate the idea. The class C manages the configuration data. The method *getString* fetches values of String-type options in the class. We first mark this method as an API of getting option values. Then all

A configuration class

```
1. class C {
2.   ...
3.   //fetch an option value
4.   String getString(String key){
5.   ...
6.   }
7. }
```

Two classes of reading option values

```
1. class P1 {
2.  private String exampleOption="p.example.path";
3.  ...
4.  void e1(C c){
5.    String path=c.getString(exampleOption);
6.    ...
7.  }
8. }
9. class P2{
10.   C config = new C();
11.   ...
12.   void test(){
13.     String version= config.getString("p.example.version");
14.     ...
15.   }
16.}
```

Figure 5.3: An example for illustrating our idea.

instances of class C are identified, i.e., instance c in the class $P1$ and $config$ in the class $P2$ respectively. The all sites of $getString$ on these instances are located. The statements where these call sites are identified as ORPs. In the example, they are statements at line 5 in the class $P1$ and at line 13 in the class $P2$. Finally, we infer which option each ORP reads. The ORP at line 13 reads a string constant, which is directly identified as the name of option being read, i.e., "p.example.version". For the ORP at line 5 in the class $P1$, variable *exampleOption* is passed as an argument. We infer the value of *exampleOption*, the option name "p.example.path", by checking the initialization statement at line 2.

5.1.3 A Challenge and Solution

The idea of identifying option read points is straightforward. We use static analysis techniques to identify call sites of marked methods and infer names of option values. The identification of call sites of marked methods relies on the usages of instances of a configuration class. An instance can be declared in a method, in a class but outside a method, or globally. The usages of an instance can span multiple methods, classes, or packages. Similarly, in the inference of an option name, a variable passed to an option read point can be declared in different methods, classes, or packages. How to identify the scope of a variable in static analysis is a major challenge for implementing our idea.

We develop an algorithm for identifying the scope of a variable. The algorithm identifies the scope of a variable based on the location of the declaration statement of the variable. For a local variable, the algorithm considers the smallest block as its scope. For an instance variable, our algorithm considers the class and its all descendant classes. For a class variable, the whole program is its scope.

5.2 Problem Statement

There are multiple configuration models used in configurable software systems in practice. According to the study [61], the key-value configuration is a common and widespread approach for users to configure applications. The configuration model is supported by the POSIX system environment, the Java Properties API, and the Window Registry. Our work targets applications with the key-value configuration model.

A configuration file

```
...
key_1 := value_1
key_2 := value_2
key_3 := value_3
...
```

A configuration class

```
...
boolean LoadFile(String Path){...}
int getInt(String keyName){...}
String getString(String keyName){...}
...
```

A class where a statement reads the value of an option

```
...
Configuraion conf=new Configuration();
String keyName = "key_1";
String var_1=conf.getString(keyName);
...
```

Figure 5.4: An example scenario of the key-value configuration schema

Key-value Configuration. The key-value configuration model can be illustrated by the example shown in Figure 5.4. Configuration options are designed as a set of key-value pairs and stored in a configuration file. The keys are strings and the values have arbitrary type. Each pair corresponds to an application attribute. Users are able to control features of applications by setting attribute values in the configuration file.

Meanwhile, application programs have a dedicated class for managing these configuration options, called a *configuration class*. The class takes responsibility of loading key-value pairs in the configuration file to a map, and offers a set of methods like *getInt* and *getString*, each of which takes one option name as a parameter and returns the value of the option. We call methods of reading option values in a configuration class as *get-methods*.

Programs read option values by calling these get-methods. In the example, *conf.getString(keyName)* returns the value of the option with name *key_1*. This statement is called as an option read point of the option named *key_1*.

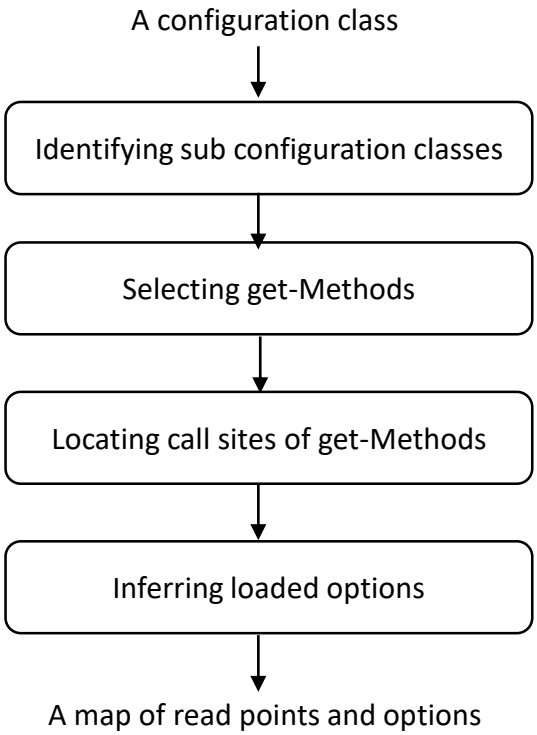

Figure 5.5: The workflow of our technique

Problem Formulation. We target programs using the key-value configuration model. Given the source code of a program and the class name of its configuration class, we attempt to automatically identify option read points, i.e., call sites of *get-methods* in the program and infer which option is loaded at each call site, finally outputting a map of option names and their read points in the source code.

5.3 ORPLocator

Our technique targets applications written in object oriented languages such as Java, C++, and C#. We abstract the source code as a set ψ of entities of classes, interfaces, and enums, which are distributed in different class files. Each entity e has a simple name and a fully-qualified name. The fully-qualified name consists of the package name and the simple name. An entity is retrieved by its fully-qualified name from ψ. Statements in classes are denoted by a tuple $\langle f, l \rangle$where f represents the name of its class file; l represents the line number of the statement in the class file.

5.3.1 Overview

Our technique requires the source code of a program and specifying its configuration class name. Its workflow is illustrated in Figure 5.5. The first step identifies subclasses of the given configuration class. The second step selects get-methods in the configuration classes. Then, all call sites of these get-methods, i.e. option read points, are located in the source code. Finally, names of the options read by each call site are inferred and a map between these option names and their read points is reported to users.

5.3.2 Identifying Subclasses of the Configuration Class

Algorithm 1 Finding subclasses of a base configuration class

Input: the source code ψ and a name of a base configuration class C
Output: S = the set of names of subclasses of C

```
1. workList ← {C}
2. while (workList ≠ ∅ ){
3.     o ← remove the first node of workList
4.     foreach (entity e ϵ ψ){
5.         if (e inherits class o){
6.             name ← get the fully-qualified name of e
7.             add name in workList in the end
8.             add name to S
9.         }
10.    }
11. }
12. return S
```

Modern, non-trivial applications typically have a base configuration class C dedicated to deal with configuration options. Furthermore, different components or subprograms of the application have its own configuration classes obtained by extending or inheriting from C. In order to obtain all call sites of get-methods in the program, we need to find all such subclasses and store them in a set S.

Given a base configuration class with a fully-qualified name C, we identify all its subclasses by Algorithm 1. In this algorithm, the simple name of a class is extracted by removing its package name from the fully-qualified name. If the declaration of an entity in ψ uses the keyword **extends**, which is followed by the simple name of the class, the entity is considered as a subclass of the class. Its fully-qualified name is computed by combining the simple name and the package name of the entity.

5.3.3 Identifying the Get-Methods

Obviously, not all methods in a configuration class are get-methods, and distinguishing get-methods from other methods is necessary. Rabkin and Katz observe that methods for accessing option values usually have a common characteristic: their names obey a naming convention, starting with the same prefix like *get* [61]. For particular types of option values, method names which reveal the returning types are given such as *getBoolean*, *getInt*, and so forth. This naming convention for configuration APIs holds in many programs. In our prototype, we adopt this convention and consider the methods in the configuration class whose names start with prefix *get* as get-methods.

The naming convention of methods accessing option values may not hold in some programs. In these cases, we need to manually check each method in the configuration class and select get-methods.

5.3.4 Locating Call Sites of Get-Methods

Intuitively, one can obtain call sites of a method from a specified class by directly searching the method name in the source code. These search results are inaccurate and would contain call sites of methods which have the same name from different classes. In order to accurately locate call sites of get-methods, we first identify instances of configuration classes (Section 5.3.4) and then locate call sites of get-methods of these instances (Section 5.3.4).

Identifying Instances of a Configuration Class

We identify instances of a configuration class by variables declared with the type of the configuration class as well as scopes of these variables. An instance is represented by a tuple $\langle v,\ s\rangle$, where v is the name of a variable and s is the scope of the variable. All instance variables of configuration classes are stored into a set V.

For any entity $e\ \epsilon\ \psi$, we check each statement in e. If a statement is a declaration statement and the declared type is one of configuration classes in the set S, the declared variable v is considered a variable of configuration classes.

The scope of a variable is determined based on three cases. First, the scope of an instance variable or class variable is identified as the largest block of the class. Second, if the variable is a formal parameter of a method, its scope is the corresponding

method. Last, the variable is declared locally. We consider the smallest block which contains the declaration statement of the variable as its scope.For the case that the variable is defined in the condition statement of a loop statement, the block of the loop statement is considered as the scope of the variable. For instance, $for(inti = 0; i < 10; i++)\{...\}$, the scope of variable i is the loop body.

Searching Call Sites of Get-Methods

⟨methodCall⟩ → ⟨methodName ⟩(⟨argumentList ⟩) |
⟨reference⟩⟨selectionOperator⟩⟨methodName⟩(⟨argumentList⟩)
⟨selectionOperator⟩→ ⟨Operator⟩
⟨reference⟩ → ⟨expression⟩
⟨methodName⟩ → ⟨identifier⟩
...

Figure 5.6: A segment of Backus-Naur Form (BNF) grammar specification for a method call

Once the configuration class instances and the corresponding scopes are identified (Section 5.3.4), we locate call sites of get-methods referenced by these variables in their scopes.

The grammar of a method call is shown in Figure 5.6. Based on this grammar, we classify method calls of get-methods into three categories. First, the *⟨reference⟩* in the grammar refers to an instance variable of a configuration class. For any variable $\langle v, s\rangle \epsilon\ V$, we search all statements in the scope s of the variable and identify method calls which have the pattern *⟨v⟩⟨selectionOperator⟩⟨methodName⟩,* where the method name can be any one of the get-method names. These call sites are stored in a set Ω.

Second, the *⟨reference⟩* refers to an instance returned by another method call. We adopt a conservative solution to deal with this case. All methods which return configuration class types are identified. By these names, we search the whole program and obtain call sites of these methods. If these call sites are followed by *⟨selection-Operator⟩⟨methodName⟩*, where the method name can be any one of the get-method names, we consider them as call sites of the get-methods and add them to Ω.

Last, inside configuration class, the *⟨reference⟩* can be implicit. For instance, the keyword *this* is used to reference to the object typed as the current class in Java. Even some get-methods are called without the reference. In order not to miss such

call sites, we identify all call sites of get-methods in a configuration class and append them to the set Ω.

5.3.5 Inferring Option Names

This section describes how we infer the name of an option read at a specific option read point. Based on this knowledge, a map between option names and their read points can be created.

In key-value configuration model, a specific option name is passed to a get-method through its call site and this site returns the value of this option. As stated in Section 5.1.1, call sites of get-methods usually take a variable storing an option name as a parameter instead of a string constant. We have to track down the value of the actual variable in a call site and obtain the option name.

Our investigation shows that variables storing the option names have characteristics which distinguish them from variables carrying other values. Variables with option names are typically initialized when they are declared and not reassigned by new values before being read. Values of such variables can be obtained by searching their declaration statements and scanning their initial values. This heuristic was used in several past papers [61, 14]. Besides, their initial values are often not string constants yet expressions combining a variable and a string constant or other variables. In the example in Figure 5.2, variable D is initialized by variable C and a string constant ".combined" and variable C is initialized by variable B and a string constant ".providers". This usage creates convenience for managing options for different components of a program.

For this complex usage of configuration options, we implement two distinct approaches to track down which option (identified by name) is read at a call site: identifying declaration statements of variables (Section 5.3.5) and computing variable values (Section 5.3.5).

Identifying Declaration Statements of Variables

The usage of variables is represented by the grammar in Figure 5.7. As the grammar shows, variables can be accessed in two ways.

Direct variable names. A variable which can be accessed directly by its name could be a local variable, an instance variable, a class variable, or a parameter variable.

⟨variableUse⟩ → ⟨variableName⟩ | ⟨reference⟩⟨selectionOperator⟩⟨variableName⟩
⟨selectionOperator⟩→ ⟨Operator⟩
⟨reference⟩ →⟨expression⟩
⟨variableName⟩ → ⟨identifier⟩
...

Figure 5.7: A segment of Backus-Naur Form (BNF) grammar specifying the use of a variable

Algorithm 2 Finding the declaration statement of a variable without a reference

Auxiliary functions:
searchDeclAsClassFields(var, o) : search the declaration statement of variable *var* among the declaration statements in the field of class *o* and superclasses of class *o*

Input: the statement of accessing a variable *var* and the current class $o \in \psi$
Output: the declaration statement of the variable
locateDeclNoReference(*var*, *o*)
1. **if**(*searchDeclInLocal (var)*)
2. **return** the matched statement
3. **if**(*searchDeclAsParameters(var)*)
4. **return** *null*
5. **if**(*searchDeclAsClassFields(var)*)
6. **return** the matched statement
7. **if**(*searchDeclAsImportedVars(var)*){
8. locate the class o' where the variable is declared
9. **if**(o' not in ψ)
10. **return** *null*
11. **if**(*searchDeclAsClassFields(var, o′)*)
12. **return** the matched statement
13.}

We locate variable's declaration statements based on this type of the declaration statement in the class.

First, the variable is considered as a local variable. We search the declaration statement of this variable in the block where the variable is used. If this is not successful, the search is extended to the outer block until the largest block of the method is reached.

If the declaration statement of the variable is not found in the method, the variable is considered as an instance variable or class variable. We search its declaration statement in the class where the variable is used but outside of any methods in the class. A variable in a class might come from its super classes. If we fail to obtain

the declaration statement in the current class, we repeat the search on field members of its super classes (if they are defined in the program, i.e. not from the library or third-party packages).

If the declaration statement still is not located, the variable is considered imported from other classes. The imported variable can be used without specifying the class in which the variable is defined. For instances, the keywords *import static* is used to import a class variable in Java. Our technique also considers this usage of a variable by matching the imported class variables. If the variable is imported, the fully-qualified name of the class where the variable is defined is extracted from the full name of the imported variable. The entity of the class can be selected by retrieving its name from ψ if the class is not from the library or the third-party packages. Then we search the declaration statement of the variable in this class.

The algorithm for extracting the declaration statement of a variable without a reference is shown in Algorithm 2.

Variable names with references. An instance or class variable can be accessed with a reference. The syntax of accessing those variables is like *⟨reference⟩⟨selectionOperator⟩ ⟨variableName⟩*. Our static analysis considers three cases of this usage. First, the reference is keyword *this* referencing the current instance or class. We search the declaration statement of the variable in the field of the current class. Second, the reference is a class name and the variable is a class variable. We locate the class this reference represents, in which the declaration statement of the variable is searched. Last, the reference is a variable name and the variable is a class member. For this case, the declaration statement of the reference variable is first located and the data type of the reference variable is obtained. If the data type is defined in the application, we select the entity of this data type and search the declaration statement of the class member in this entity.

We designed Algorithm 3 for both usage cases: direct variable names and variable names with references. If a variable is accessed directly by its name, we invoke Algorithm 2. For the usage of a variable with a reference, the algorithm considers the reference of the variable as an instance or class variable. If the reference is other expression like the call site of a method, *null* is returned. In the case where the type of the object of the reference is not defined in the application, *null* is returned too. Similarly, Algorithm 2 searches super classes of a class for locating the declaration statement of an instance or class variable.

Algorithm 3 Locating the declaration statement of a variable

Input: the statement with access to a variable *var* and the class $o \in \psi$
Output: the declaration statement of the variable
locateDecl(*var*, *o*)

```
1.if(var without reference)
2.   return locateDeclNoReference(var, o)
3. else{
4.        reference ← getReference(var)
5.        if(reference is a key word this)
6.              return searchDeclAsClassFields(var, o)
7.       o' ←locateClassEntity(reference)
8.        if(o' is not null)
9.              return searchDeclAsClassFields(var, o')
10.      declStat ← locateDeclNoReference(reference, o)
11.       if(declStat is not null){
12.             type ← getType(declStat)
13.             o' = locateClassEntity(type)
14.              if(o' is in ψ)
15.             return searchDeclAsClassFields(var, o')
16.       }
16.       return null
17. }
```

Computing Values of Actual Parameters

As stated above, in many cases parameters of an option read point are initialized by an expression combining a variable and a string constant or a variable expression. These expressions can be modeled by the grammar shown in Figure 5.8.

```
⟨expression⟩ → ⟨S⟩| ⟨S⟩+ ⟨S⟩
⟨S⟩          → ⟨variable⟩ | ⟨stringLiteral⟩
...
```

Figure 5.8: A segment of Backus-Naur Form (BNF) grammar specification for expressions of generating an option name

In order to infer which option name is used by an option read point, we propose Algorithm 4. First, the algorithm obtains operands and operators of an expression *expr*. There are two cases for an operand in this model. If the operand is a string literal, its value is stored into *str*. If the operand is a variable, we call Algorithm 3 to locate the declaration statement of the variable and obtain its initial expression *expr'*. Then we recursively call Algorithm 4 to compute the value of *expr'* until the

Algorithm 4 Inferring values of actual parameters at a call site

```
Input: an expression expr
Output: a string literal str
   inferValue(expr)
1. optionName ← null
2. elements ←getElements(expr)
3. for (element element in elements){
4.     if(element is an operand){
5.         if(element is a string literal)
6.             optionName← element
7.         if(element is a variable){
8.             currentClass ←getCurrentClass(expr)
9.             declStat ←locateDecl(element, currentClass)
10.            expr′ ← getInitializedExpression(declStat)
11.            optionName ← optionName + inferValue(expr′)
12.         }
13.         else
14.             return null
15.    }
16.    else if(element is an operator "+")
17.        continue:
18.    else
19.        return null
20. }
21. return optionName
```

initial expression of a variable is a string literal. If the values of all operands are successfully inferred, the combination of values of all operands is returned. Note that our technique does not consider expressions which do not follow the model in Figure 5.8. Our evaluation shows that this algorithm can infer values of most variables except when the value of a variable is generated by another method.

According to our experience, most of the time, option names are concatenated by using the operator "+" instead of calling APIs for concatenating strings. Consequently, in Figure 5.8, we only consider the operator "+".

In the end, a map is built between option names and the corresponding option read points. By searching for option names in the documentation we can obtain the map between documented options and their read points in the program.

5.3.6 An Example

We use the example in Figure 5.2 to illustrate how our technique infers the option names used by option read points. Here a call site *conf.getBoolean(D, true)* is located at line 116 in the class file *CompositeGroupMapping.java.* The call site takes variable D storing an option name as a parameter. The goal of our technique is to infer the value of variable D.

Algorithm 4 takes variable D as input and considers it as a variable instead of a string constant. Then Algorithm 3 is called to identify the declaration statement of variable D. Since variable D is accessed directly by variable name, Algorithm 2 is called to identify its statement at line 48 in the class file *CompositeGroupMapping.java* and returns the initial expression of the statement C + *".combined*". Algorithm 4 parses the expression. The string constant ".combined" is stored in a variable *optionName*. Then Algorithm 4 starts to infer the value of variable C in the expression. Similarly, Algorithm 4 obtains B + *".providers*", the initial expression of variable C. Then the string ".providers" and ".combined" is combined to ".providers.combined" and stored in variable *optionName.* Inference of the value of variable B is started.

When locating the declaration statement of variable B, Algorithm 3 finds out that variable B is not defined in the class *CompositeGroupMapping.* Then the algorithm identifies the super class of class *CompositeGroupMapping,* i.e. *GroupMappingServiceProvider,* where the declaration statement of variable B is found and its initial expression *CommonConfigurationKeysPublic.A* is returned to Algorithm 4. Algorithm *4* continues to infer the value of variable A. While locating the declaration statement of variable A, Algorithm 3 finds variable A is accessed with a reference *CommonConfigurationKeysPublic* which is a class name. Algorithm 3 locates class *CommonConfigurationKeysPublic,* where the declaration statement of variable A is found and its initial expression hadoop.security.service.user.name.key is returned to Algorithm 4. Algorithm 4 finally outputs the value of variable D, i.e., hadoop.security.service.user.name.key. providers.combined.

Summary. This section presents our approach, ORPLocator. Given source code of a program and its configuration class, ORPLocator first identifies all classes which extends the input configuration class and marks methods fetching option values. Then ORPLocator identifies instances of all configuration classes and locates the call sites of marked methods on these instances. These call sites are option read points in the source code. For each call site, we infer which option is loaded by computing the values of parameters passed into the call site. Finally, ORPLocator outputs a

map between names of all options which are used by the program and their read points in the source code. With this map, inconsistencies between source code and configuration documents can be detected.

5.4 Implementation

We implemented our technique as a prototype, called Option Read Points Locator (ORPLocator), which is restricted to applications in Java. The tool relies on the srcML library [69]. The library computes an XML representation for source code, where the markup tags identify elements of the abstract syntax for the language. Currently srcML supports mainstream programming languages such as Java, C++, C#, and C. Our analysis targets applications in written multiple languages. Consequently we choose srcML instead of Java analysis tools such as JDT [3] and Spoon [7].

In the srcML format, all elements in the source code are wrapped with XML elements, which allows leveraging XML tools to extract various information by textual search using regular expressions. By employing XPath [86] in ORPLocator, we are able to query all entities, i.e. , classes, interfaces and enums, retrieve an entity by its fully-qualified name, and so on.

srcML creates a high-level intermediate program representation. Operations on data in this format cost a lot in terms of time and space. In order to optimize performance, ORPLocator builds a map between XML nodes of all entities and their corresponding fully-qualified names. The retrieval of an entity by its fully-qualified name does not require scanning the whole srcML representation of a program.

ORPLocator adopts Dom4J [23] as a parser which allows flexible queries on XML nodes in the srcML representation.

5.5 Evaluation

We conducted an empirical study to evaluate the effectiveness and usefulness of ORPLocator and attempted to answer the following research questions.

- RQ1: How effective is ORPLocator in locating option read points and identifying option names?
- RQ2: Are options in the document of a module read by the module?

Table 5.1: Subject programs.

Modules	#Java Files	#LOC	#Options
Common (2.7.1)	1,495	294,898	127
HDFS (2.7.1)	1,380	400,353	216
MapReduce (2.7.1)	1,275	255,670	172
YARN (2.7.1)	1,612	354,901	197
Summary	5,762	1,305,822	712

- RQ3: How does ORPLocator's effectiveness compare to existing techniques?

- RQ4: What is the time cost of locating option read points by ORPLocator?

5.5.1 Experimental Setup

Subject Programs

We evaluated ORPLocator on the Apache Hadoop framework [34] for scalable and distributed computing, which mainly consists of four modules shown in Table 5.1. In the table, Column "#Java Files" is the number of Java files. Column "#LOC" is the number of lines of code. They are counted by CLOC [19]. Column "#Options" is the number of documented options for each module. Aside from these 4 modules, Hadoop consists of more than 10 tools. In the evaluation, we do not consider these tools because they have few configuration options.

There are considerations for choosing the latest version of Hadoop (version 2.7.1) as the subject program. First, Hadoop is currently widely used to store, analyze and access large amount of data and has evolved into a complex ecosystem with more than 100 related systems. The configuration mechanism in Hadoop has matured through the evolution across a number of versions from 0.14.1 to 2.7.1. Choosing Hadoop as the subject program is representative. Second, Hadoop has an abundance of configuration options.

Evaluation Procedure

The four modules are independently implemented and each of them has a configuration file which contains its own options. In order to accurately evaluate our technique, we consider each module as a single program.

Table 5.2: The results of ORPLocator.

Modules	get-Method Callsites	Detected by ORPLocator		Documented	
		#Options	#ORPs	#Options'	#ORPs'
Common	352	210	261	109	147
HDFS	586	367	524	206	335
MapReduce	909	423	631	162	259
YARN	701	300	445	181	322
Summary	2548	1300	1861	658	1063

For each module its source code (excluding the test code) is converted into the srcML format by using the tool srcML [69]. Then we provide the resulting srcML file as well as the name of the configuration class as input to our tool ORPLocator. Finally, the ORPLocator outputs a map between option names and their read points for each module.

5.5.2 RQ1: Effectiveness

We evaluate the effectiveness of ORPLocator by computing the number of identified options out of documented options and checking if a located option read point does read an option value in the program. The correctness of a located option read point is manually checked by inspecting source code. In the evaluation, two people performed the manual check and resolved discrepancies. Although this is not a guarantee of correctness, it excludes the presence of obvious issues with the manual check.

Results

The results obtained by ORPLocator are shown in Table 5.2. Column "Modules" lists modules of Hadoop we studied. Column "get-method Callsites" shows the number of call sites of get-methods located by ORPLocator. Column "Detected by ORPLocator" provides the output of ORPLocator. Column "#Options" and "#ORPs" show the number of detected options and the number of their read points, respectively. Column "Documented" reports how many of the detected options are documented. Column "#Options" displays the number of options which are detected by ORPLocator and Column "#ORPs" shows the number of their read points.

As shown in Table 5.2, ORPLocator successfully locates 1861 read points for 1300 options (i.e., distinct option names) in the source code of the 4 modules. From these

options, 658 are documented. The number of option read points is larger than the number of options because one option might have multiple read points in the program.

Not all call sites of get-methods are option read points (as we can see in Table 5.2). We manually checked these call sites which are not treated as option read points by ORPLocator. They can be divided into two cases.

First, the get-methods of these call sites are not methods which load values of configuration options from configuration files. For instance, the class *JobConf* in MapReduce extending the configuration class have many get-methods which are not for producing option values such as *getKeepTaskFilesPattern()*, *getNumReduceTasks()*, and so on.

Second, the get-methods of these call sites are methods loading option values from configuration files. But the names of options loaded by these call sites are generated by methods. They cannot be inferred by our technique. For instance, in the call site *conf.get(hadoop.rpc.socket. factory.class. + clazz.getSimpleName())*, a part of the option name is generated by another method. The statistic of our result data shows that luckily this kind of option usages is quite rare.

Third, the call sites do not call get-methods. They are introduced by the conservative solution we take in Section 5.3.4. In the conservative solution, for a method returning a configuration class type, we search its method name in the whole program and get call sites. Some of these call sites do not return an instance of a configuration class because they invoke different methods which have the same method names but do not return a configuration class type. These call sites do not load the values of configuration options.

From the data in Table 5.2 we can see the number of options detected by ORPLocator is larger than that of documented options. Manual checking shows they are indeed used to control the behavior of the programs by changing values of these options. Part of them are documented for end users. As for options missed by ORPLocator, we conduct a further investigation in the following section.

Options Whose Read Points Are Not Located

In this section, we conduct an study on documented options whose read points are not located by ORPLocator. We break down these options into 6 categories by investigating how the options are used in the source code (see Table 5.3).

Table 5.3: Categories of options whose read points are located.

Categories	Description	Common	HDFS	MapReduce	Yarn	Sum
1	Their read points are located in other modules	11	0	0	2	13
2	Option names are deprecated in the version	0	1	1	0	2
3	Option values are loaded by other configuration classes	4	0	0	0	4
4	Option names are mistakenly documented	1	1	2	0	4
5	Option names are determined in runtime	2	7	7	13	29
6	Call sites reading values of these options are missed	0	1	0	1	2
The number of un-located documented options for each module		18	10	10	16	54

Table 5.4: Bugs detected by ORPLocator

Issue	Type	Status	Documented option names	Option names read in the source code
HADOOP-12704[1]	Bug	Resolved	hadoop.**work.around**.non.threadsafe.getpwuid	hadoop.**workaround**.non.threadsafe.getpwuid
MAPREDUCE-6605[2]	Bug	Resolved	mapreduce.map.skip.**proc.count.autoincr** mapreduce.reduce.skip.**proc.count.autoincr**	mapreduce.map.skip.**proc-count.auto-incr** mapreduce.reduce.skip.**proc-count.auto-incr**
HDFS-8274	Bug	Resolved	nfs.dump.dir	nfs.**file**.dump.dir

[1] https://issues.apache.org/jira/browse/HADOOP-12704
[2] https://issues.apache.org/jira/browse/MAPREDUCE-6605

Each module in Hadoop has its own configuration file. An option of a module might be inserted into configuration files of other modules. Also a read point of an option in a module may exist in other modules. Category 1 indicates options of a module whose read points are located in other modules. For Hadoop Common, there are 11 documented options which are unread by Common but read by HDFS or Yarn. Similarly, 2 of options in Yarn are only read by MapReduce.

Category 2 represents options which are deprecated in the current version, but are still not removed from the list of documented options. There are two deprecated options: "nfs.allow.insecure.ports" for HDFS and "mapreduce.job.counters.limit" for MapReduce. The read points of their corresponding new options are successfully located by ORPLocator.

Category 3 indicates options which are loaded by other configuration classes instead of the main configuration class of Hadoop. There are 4 options for Common loaded by the Properties class in Java.

Category 4 shows options whose names are erroneously documented. We have reported these bugs to developers, which are confirmed and fixed. The reported bugs are shown in Table 5.4.

Category 5 represents options whose names or part of names are generated by another method. For instance, in the call site *conf.get(hadoop.rpc.socket.factory.class. + clazz.getSimpleName()),* the last part of the option name is generated by the method *getSimpleName()*. ORPLocator failed to infer names of these options based on their read points. This is one of the limitations of ORPLocator.

Category 6 displays options which are read in the program but whose read points were not located by ORPLocator. The reason is that ORPLocator failed to identify call sites of get-methods which read values of these options. We further discuss it in Section 5.5.6.

Summary of RQ1. Although our technique bases findings from empirical studies [61, 14], the evaluation shows that ORPLocator is effective in locating option read points. It successfully detects 1861 read points of 1300 distinct options for 4 modules we studied. For documented options, it locates at least one read point for 120 (109 + 11) out of 127 (94%) options in Common, 206 out of 216 (95%) options in HDFS, 164 (162+2) out of 172 (95%) options in MapReduce, 181 out of 197 (93%) options in Yarn. Note we consider options whose read points are located in other modules.

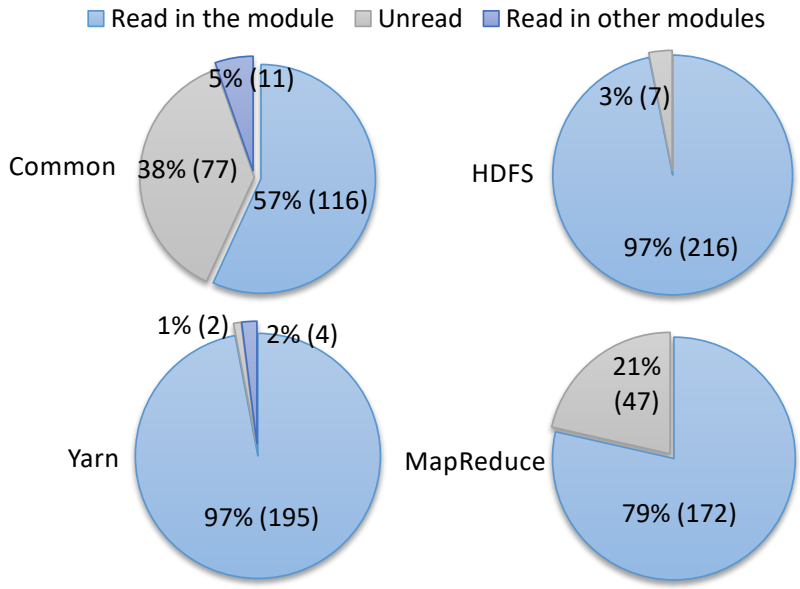

Figure 5.9: The distribution of read points of documented options for each module

5.5.3 RQ2: Option Inconsistencies

Hadoop consists of four main modules, which have their own documents of configuration options. Over the recent years, each module has undergone a significant development. The number of configuration options for each module increased to more than 200. Are all options in the document of a module read by the module?

The study assumes that the value of an option is read by a program if the option has at least one read points using option's name. For the options on which ORPLocator failed, we inspected the source code to check whether they are read by the program.

The result is shown in Figure 5.9. For each module, the options are broken into three categories: read in the module, not read in the 4 modules we studied, and read in other modules. As we can see, for Common and MapReduce a significant part of options in the default configuration file are not read by the module itself. The primary reason is that options of many Hadoop ecosystems or components are put in the documents of these two modules.

Some names of documented options are not correct. As shown in Table 5.3, we found that 4 option names are wrongly documented in the released configuration files and 2 option names are deprecated in the current version. We submitted the 4

Table 5.5: The number of entry points for each module.

Modules	Common	HDFS	MapReduce	Yarn
# Entry points	22	25	8	13

wrongly documented options to Hadoop developers. They confirmed these issues and will release patches in the next version.

Summary of RQ2. Not all options in the document of a module are read by the module. Within options in the document of Common, only 57% are read by Common. MapReduce reads 79% of documented options. Additionally, 4 incorrect option names are found among the documented options; also 2 deprecated option names were found. HDFS and Yarn nearly read 97% of their documented options.

5.5.4 RQ3: Comparison with a Previous Technique

This section compares our technique with a previous technique, called Confalyzer [61], which is recently used by SCIC [14] to check software configuration inconsistencies. Within our best knowledge, Confalyzer is the most precise technique of locating option read points known in the literature.

Confalyzer, proposed by Rabkin and Katz [61], is a tool of extracting program configuration options assuming the key-value model. The core idea of Confalyzer is similar to ours and considers methods starting with *get* in the configuration class as APIs accessing option values. Then it identifies where these methods are called in the program by building a call graph and finds string parameters at these call sites, taking these parameters as options and call sites as option read points.

Running Confalyzer. The tool is published on GitHub[3]. For running it one needs to specify the entry points of analyzed programs. Missing entry points would decrease accuracy of detected results. To make an end-to-end comparison, we identified all possible entry points of each module by searching main methods in the source code (see Table 5.5). Confalyzer also takes the Properties class in Java as a configuration class of Hadoop, which is not considered by ORPLocator. The options loaded by the Properties class are not considered.

Results. We collected all distinct options and their read points reported by Confalyzer and selected options which are documented (as well as corresponding read points). The results are shown in Figures 5.10 and 5.11. We can see that ORPLocator

[3]https://github.com/asrabkin/Confalyzer

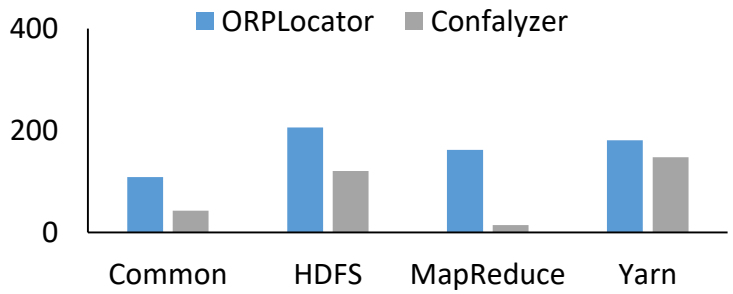

Figure 5.10: The comparison on the number of documented options

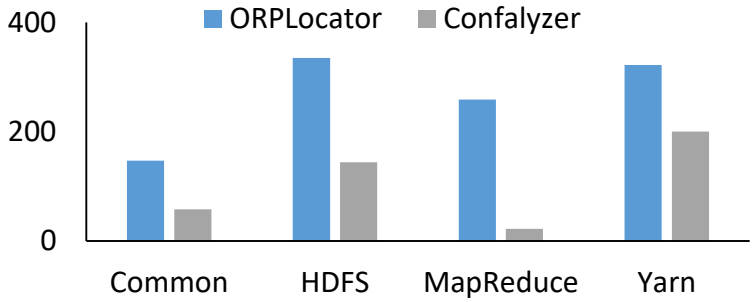

Figure 5.11: The comparison on the number of read points of documented options

detects significantly more documented options and corresponding option read points than Confalyzer.

ORPLocator is more accurate than Confalyzer primarily for three reasons. First, ORPLocator detects option read points by scanning the *whole* source code to match call sites of configuration APIs. Contrary to this, Confalyzer identifies option read points by constructing a call graph using static analysis. Hadoop heavily uses reflection and this may cause incompleteness in the inferred call graph [48, 60]. Some methods containing option read points might not be included in the call graph. Second, ORPLocator considers subclasses of the base configuration class, which helps identify get-methods added in the configuration class for sub programs. Third, Confalyzer acquires option names by reading the value of actual parameters at call sites if the actual parameters are compile-time constants. Its accuracy depends on compiler optimization. ORPLocator implements a simple parser to infer values of actual parameters at call sites without this limitation.

Compared to Confalyzer, ORPLocator has another two significant advantages. First, Confalyzer requires a complete application program. Otherwise, the error due

to missing classes propagates during building a call graph. We encountered this error many times when running experiments. Second, Confalyzer needs all entry points of a program. ORPLocator has none of these issues.

Summary of RQ3. ORPLocator produces more accurate results in locating option read points and does not require entry points of analyzed programs.

5.5.5 RQ4: Time Cost

Table 5.6: The analysis time and file sizes of srcML output.

Modules	Common	HDFS	MapReduce	Yarn
File size (mb)	65.6	86.9	57.8	90.7
Time (min)	69.9	135.7	90.0	172.2

ORPLocator uses srcML output as the intermediate representation, which is a high-level and expensive representation compared to low-level representations such as SSA [21] or LLVM [45]. The performance is a crucial issue for ORPLocator. This section evaluates the time overhead of locating option read points.

Our experiments were conducted on a laptop with Intel i7-2760QM CPU (2.40GHz) and 8 GB physical memory, running Windows 10. The analysis time and the size of srcML files for each module are shown in Table 5.6.

Summary of RQ4. As we can see, our analysis needs 1 to 3 hours for each module. This is substantial yet acceptable considering that the scale of each module is quite large, and an analysis is performed infrequently.

5.5.6 Discussion

Limitations

Our technique of locating option read points has some limitations. First, as we explained in Section 5.3, ORPLocator focuses on configuration of the key-value style, with methods accessing option values have names starting with *get*, otherwise we have to manually select them. Second, the accuracy of the identification of option read points relies on the patterns how the get-methods are called in the program. The implicit invocation of get-methods will be missed by our technique, for instances, if get-methods are called by a complex call chain. Instances of configuration classes are stored in the complex data structures like a array list. Last, our technique assumes

most of the option names are not generated by a method. Otherwise, the option names fail to be inferred.

Threats to Validity

The primary threat to external validity of this work concerns whether or not our results will generalize. The access techniques to configuration option values varies in different programs. This include the construction of option names and the invocation patterns of configuration APIs. This variation may affect effectiveness of our technique. To mitigate this threat, we used Apache Hadoop, a Java program with one of the largest number of configuration options of the open-source projects. Moreover, it is a program which is developed by a variety of developers and widely used in both academy and industry. Although we need to conduct more evaluations on different programs, we believe that our results can generalize to other programs in Java which uses key-value style configuration. In the future work, we plan to instantiate our technique for other programs.

One internal validity threat regards the identification of option read points. In our technique, all call sites of get-methods in configuration classes are considered possible option read points. Some call sites of methods which are not configuration APIs are interpreted as option read points. To mitigate this threat, we manually check all read points of documented options in Hadoop Common. The results show all of them are option read points without false positives. Sure, our manual inspection of code inspection might be error-prone. Finally, the data of option read points produced by ORPLocator and Confalyzer is all available online[4].

5.6 Summary

This chapter addresses configuration errors due to inconsistencies between configuration documentation and source code. We present a practical technique, ORPLocator, which is capable of building a map between names of configuration options and their read points in the source code. ORPLocator adopts a static analysis technique to identify option read points in the program and infer the names of the options read there. Compared to existing techniques, our analysis computes dependency information as needed without building a program dependency graph or system dependency graph.

[4]https://goo.gl/7uVzYZ

We conduct an empirical study on the latest version (2.7.1) of Apache Hadoop, a widely popular framework for distributed data processing with more than 1.3 million lines of source code and 800+ configuration options. Results show that ORPLocator is effective in identifying option read points in source code. ORPLocator is able to successfully locate at least one read point for 93% to 96% of documented options within four Hadoop components. We compare ORPLocator with a previous state-of-the-art technique. The experiment shows that ORPLocator produces more accurate results. Our analysis needs 1 to 3 hours for each module.

By comparing extracted options with documented options for each module, we find that a significant part of documented options are not read by the module. Besides, our experimental study discovers 4 previously unknown inconsistencies between documented options and source code.

Chapter 6

Conclusion

This thesis has presented two applications of static program analysis to debug software configuration errors. One application addresses misconfigurations due to mistakes in option values. The other addresses misconfigurations due to inconsistencies between configuration documentation and source code. In the next sections, we summarize the major technical contributions of our work. Following that, we outline some possible avenues for future work.

6.1 Summary

Chapter 4 presents a technique, automated diagnosis of software misconfigurations via static analysis, and evaluates the technique on 29 configuration errors from 4 real-world applications. From this work, we can draw the following conclusions.

Static analysis can infer whether an option value flows to the crashing site of an error at runtime. Our experimental evaluation shows that there are dependencies between the values of root-cause configuration options and the crashing sites for almost configuration errors we collected. The data flow of an option value to the crashing site can be inferred via static analysis.

The heuristic recently-options-read can be used for misconfiguration debugging. We believe the recently read options before failing are highly likely the root cause of the error. Based on this heuristic, we propose a model for computing the correlation degree of an option with a configuration error. Experimental results show that, under this model, the root cause is ranked first in the diagnosis results for most of configuration errors.

Chapter 5 presents a static analysis technique of identifying option read points and extracting option names, which is capable of diagnosing inconsistencies between configuration documentation and source code. We evaluate the technique on 4 modules of the latest version of Apache Hadoop (2.7.1). Some conclusions are drawn from this work.

Extraction of options from programs is an effective way to diagnose inconsistencies between source code and configuration documentation. Our analysis shows a significant part of documented options are not read by the corresponding version of programs, at least for applications we evaluated in the experiment. We also find several options which are mistakenly documented and confirmed by developers after reporting these issues to them.

SrcML-based analysis is effective in identifying option read points. Based on the srcML representative, our analysis locates instances of a class and call sites of methods on a specific instance by searching their names in the corresponding scopes. Compared to the techniques based on Points-to and call graphs, our analysis achieves a significantly improvement on the accuracy of diagnosis results.

Configuration option names are available statically. Our analysis extracts option names by computing the values of parameters passed to option read points. The computation assumes variables storing option names are initialized in their declaration statements and not changed any more. The high accuracy of our analysis shows option names are available statically.

6.2 Future Directions

6.2.1 Generalizing the Problem

Our work addresses crashing errors due to mistakes in option values. A potential topic for future work would be generalizing the problem. The problem can be generalized in two aspects. First, configuration errors due to violating constraints among multiple options will be explored. The setting of an option might be dependent on one another. Violating constraints between multiple options can lead to a configuration error. For this kind of configuration errors, the root cause is not a unique option and could be a set of options. Such configuration errors can be diagnosed by extending our analysis since our current work can identify options which have data dependence with the crashing site of an error.

Second, our current analysis focuses on crashing errors due to misconfiguration. There exist many other kinds of errors due to misconfiguration, e.g., performance problems, silent failures, and memory leaks. In these cases, there is no crashing sites available. Our analysis needs to be redesigned to address these kinds of problems. Dynamic analysis is a better option since these problems are workload-dependent.

6.2.2 Misconfiguration Repair

Configuration errors have been increasing in the share of user reports. Solving these issues is costly in the administration of software systems. Automated repair of configuration errors will attract attentions in the future.

Misconfiguration repair is typically instance-specific and environment-specific. The repair of a configuration error properly works in a single instance and cannot be applied to the configuration error in different instances. The repair of a configuration error varies in different instances since the system environment or configuration setting is instance-specific.

Misconfiguration can be mitigated by improving the reaction of systems to misconfiguration. Here, reaction means how systems behave when misconfiguration occurs. Silence, hanging, or crashing without proper hint messages are bad reactions of a system to misconfiguration. Pinpointing possible root causes, i.e., wrongly-set options, would be a user-friendly reaction to misconfiguration for a system. With explicit messages, users can possibly repair configuration errors themselves without reporting them.

We believe that improving the reaction of a system to misconfiguration is an effective way to repair configuration errors. By testing systems with various configuration setting, the "bad reactions" can be triggered. For these cases, automated repair can make systems pinpoint possible root causes for a configuration error. Improving reactions of a system to misconfiguration will be an interesting direction.

6.2.3 Applying Precise Analysis to Large Scale Programs

One of the challenges for troubleshooting configuration is the huge number of options of a system. The scale of such systems is usually large. In thesis, we embrace imprecise analysis for large-scale systems. By using heuristics, our analysis achieves a high precision in the diagnosis of configuration errors. But precise analysis is able to obtain more accurate facts that how an option value has influence in the behavior of

a system. Accurate facts are the foundation to infer the root cause of a configuration error.

Path-sensitive data flow analysis on large-scale software systems is worth being explored for misconfiguration diagnosis. This analysis extracts the data flow of an option value in possible execution paths and can precisely explain how the value leads to an error.

Exploiting models for heap dependence analysis can help produce the complete data flow of an option value. Option values can flow into various collections, which can terminate the data flow if the analysis does not consider heap dependence. Analysis considering heap dependence is very expensive and cannot scale to large programs. An appropriate model for heap analysis needs to be exploited for diagnosing configuration errors in large-scale systems.

Bibliography

[1] Firefox. https://www.mozilla.org.

[2] http. https://httpd.apache.org/.

[3] Jdt. http://www.eclipse.org/jdt/.

[4] log4jcon. https://github.com/janinko/Log4J-configuration-converter.

[5] mysql. https://www.mysql.com/.

[6] Postgresql. https://www.postgresql.org/.

[7] Spoon. http://spoon.gforge.inria.fr/.

[8] D.C. Arnold, D.H. Ahn, B.R. De Supinski, G.L. Lee, B.P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, 2007.

[9] F. A. Arshad, R. J. Krause, and S. Bagchi. Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 198–207, Nov 2013.

[10] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[11] Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 281–286, Berkeley, CA, USA, 2008. USENIX Association.

[12] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–11, Berkeley, CA, USA, 2010. USENIX Association.

[13] Rob Barrett, Eser Kandogan, Paul P. Maglio, Eben M. Haber, Leila A. Takayama, and Madhu Prabaker. Field studies of computer system administrators: Analysis of system management tools and practices. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW '04, pages 388–395, New York, NY, USA, 2004. ACM.

[14] Farnaz Behrang, Myra B. Cohen, and Alessandro Orso. Users beware: Preference inconsistencies ahead. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 295–306, New York, NY, USA, 2015. ACM.

[15] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, pages 301–310, 2010.

[17] Aaron B. Brown and David A. Patterson. To err is human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY '01*, 2001.

[18] M. Chen, A.X. Zheng, J. Lloyd, M.I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 36–43, 2004.

[19] CLOC. http://cloc.sourceforge.net/.

[20] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code a tool demonstration.

[21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[22] Susan Dart. Concepts in configuration management systems. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, SCM '91, pages 1–18, 1991.

[23] Dom4J. https://dom4j.github.io/.

[24] Zhen Dong, Artur Andrzejak, David Lo, and Diego Elias Costa. Orplocator: Identifying reading points of configuration options via static analysis. In *IEEE 27th International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, Canada, 23-27 October*, 2016.

[25] Zhen Dong, Artur Andrzejak, and Kun Shao. Practical and accurate pinpointing of configuration errors using static analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 171–180, 2015.

[26] Zhen Dong, Mohammadreza Ghanavati, and Artur Andrzejak. Automated diagnosis of software misconfigurations based on static analysis. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013 - Supplemental Proceedings*, pages 162–168, 2013.

[27] S. Duan and S. Babu. Automated diagnosis of system failures with fa. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1499–1502, 2009.

[28] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. Consistency-based diagnosis of configuration knowledge bases. In *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20-25, 2000*, pages 146–150, 2000.

[29] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object oriented software, 1995.

[31] Mohammadreza Ghanavati, Artur Andrzejak, and Zhen Dong. Scalable isolation of failure-inducing changes via version comparison. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013 - Supplemental Proceedings*, pages 150–156, 2013.

[32] Jim Gray. Why do computers stop and what can be done about it, 1985.

[33] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM.

[34] Hadoop. http://hadoop.apache.org/.

[35] Chung hao Tan. Failure diagnosis for configuration problem in storage system. http://cs229.stanford.edu/proj2005/Tan-FailureDiagnosisForConfigurationProblemInStorageSystem.pdf.

[36] HBase. http://hbase.apache.org/.

[37] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[38] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A user survey of configuration challenges in linux and ecos. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 149–155, New York, NY, USA, 2012. ACM.

[39] JChord. http://pag.gatech.edu/chord.

[40] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 215–224, New York, NY, USA, 2014. ACM.

[41] Robert Johnson. More details on today's outage. http://cc4.co/CGL, September 2010.

[42] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003.

[43] A. Kapoor. Web-to-host: Reducing total cost of ownership. Technical report, Technical Report 200503, The Tolly Group, May 2000.

[44] L. Keller, P. Upadhyaya, and G. Candea. Conferr: A tool for assessing resilience to human configuration errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 157–166, June 2008.

[45] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[46] Michelle Levesque. Fundamental issues with open source software development. *First Monday*, vol. Special Issue 2: Open Source, 2005.

[47] Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 445–456, New York, NY, USA, 2014. ACM.

[48] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

[49] James Mickens, Martin Szummer, and Dushyanth Narayanan. Snitch: Interactive decision trees for troubleshooting misconfigurations. In *Proceedings of the 2Nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, SYSML'07, pages 8:1–8:6, Berkeley, CA, USA, 2007. USENIX Association.

[50] Yi min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, and Chun Yuan. Strider: A black-box, state-based approach to change and configuration management and support. In *In Usenix LISA*, pages 159–172, 2003.

[51] S. Nadi, T. Berger, C. Kastner, and K. Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *Software Engineering, IEEE Transactions on*, 41(8):820–841, Aug 2015.

[52] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 140–151, New York, NY, USA, 2014. ACM.

[53] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[54] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[55] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, April 1984.

[56] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[57] Rahul Potharaju, Joseph Chan, Luhui Hu, Cristina Nita-Rotaru, Mingshi Wang, Liyuan Zhang, and Navendu Jain. Confseer: Leveraging customer support knowledge bases for automated misconfiguration detection. *PVLDB*, 8(12):1828–1839, 2015.

[58] Christian Prehofer. Feature-oriented programming: A fresh look at objects. pages 419–443. Springer, 1997.

[59] A. Rabkin and R.H. Katz. How hadoop clusters break. *Software, IEEE*, 30(4):88–94, July 2013.

[60] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 193–202, Washington, DC, USA, 2011. IEEE Computer Society.

[61] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 131–140, 2011.

[62] Ariel Rabkin and Randy Katz. How hadoop clusters break. *IEEE Softw.*, 30(4):88–94, July 2013.

[63] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury. Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[64] Randoop. https://code.google.com/p/randoop/.

[65] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 445–454, Cape Town, South Africa, May 2010.

[66] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, pages 11–20, New York, NY, USA, 1994. ACM.

[67] Cindy Rubio-González and Ben Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 73–80, New York, NY, USA, 2010. ACM.

[68] A. Schroter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 118–121, 2010.

[69] srcML. http://www.srcml.org/index.html.

[70] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. *SIGPLAN Not.*, 42(6):112–122, June 2007.

[71] Stack Overflow. http://stackoverflow.com/.

[72] Ya-Yunn Su and Jason Flinn. Automatically generating predicates and solutions for configuration troubleshooting. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 17–17, Berkeley, CA, USA, 2009. USENIX Association.

[73] Yevgeniy Sverdlik. Microsoft: 10 things you can do to improve your data centers. http://cc4.co/USWU, August 2012.

[74] Amazon Web Services Team. Summary of the amazon ec2 and amazon rds service disruption in the us east region. http://aws.amazon.com/message/65648/, 2011.

[75] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[76] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 131–144, New York, NY, USA, 2007. ACM.

[77] Adwait Tumbde, Matthew J. Renzelmann, and Michael M. Swift. Abstract configuration data deserves a database.

[78] WALA. http://sourceforge.net/projects/wala/.

[79] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *In OSDI*, pages 245–258, 2004.

[80] M. Wang, X. Shi, and K. Wong. Capturing expert knowledge for automated configuration fault diagnosis. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 205–208, June 2011.

[81] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[82] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[83] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.

[84] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[85] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 58–68, Piscataway, NJ, USA, 2012. IEEE Press.

[86] XPath. http://www.w3.org/TR/xpath-30/.

[87] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 244–259, 2013.

[88] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, 2011.

[89] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. *SIGARCH Comput. Archit. News*, 38(1):143–154, March 2010.

[90] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 28–28, Berkeley, CA, USA, 2011. USENIX Association.

[91] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 687–700, 2014.

[92] Sai Zhang and Michael D. Ernst. Automated diagnosis of software configuration errors. In *Proceedings of the 34th International Conference on Software Engineering*, San Francisco, CA, USA, May 22–24, 2013.

[93] Sai Zhang and Michael D. Ernst. Which configuration option should i change? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 152–163, New York, NY, USA, 2014. ACM.

[94] Sai Zhang and Michael D. Ernst. Proactive detection of inadequate diagnostic messages for software configuration errors. In *ISSTA 2015, Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 12–23, Baltimore, MD, USA, July 15–17, 2015.