Automated Diagnosis of Software Misconfigurations Based on Static Analysis

Zhen Dong, Mohammadreza Ghanavati, Artur Andrzejak
Institute of Computer Science
Heidelberg University, Germany
{zhen.dong, mohammadreza.ghanavati, artur.andrzejak}@informatik.uni-heidelberg.de

Abstract—**Software configuration settings are an effective way to customize applications. However, inconsistencies or mistakes in option values can result in a system crash and need huge time and effort to diagnose. We present a technique to identify the root causes of configuration errors. It uses static program analysis to link the misconfiguration of an application to a specific configuration option. Our technique has two prominent characteristics compared to existing approaches: it relies only on static analysis, and it does not need profiles of the application with correct configuration.**

Based on the proposed techniques, we developed a tool called ConfDebugger. We evaluated its effectiveness on 8 configuration errors in the Java program JChord. ConfDebugger successfully diagnosed 7 out of 8 errors. For 5 of them, root cause was exactly pinpointed without a false positive, and in total, the average number of false positives was only 0.5. This is better than two state-of-the-art methods, with average numbers of false positives of 1.7 and 5.7, respectively.

Index Terms—**Configuration debugging, static program analysis, thin slicing, failure-inducing chop**

I. Introduction

Configuration settings provide a proven and effective way to customize applications for different user requirements. Growing functionality and size of today's software leads to a larger number of configuration options, and more complex dependency patterns between them. This in turn increases the likelihood of introducing a misconfiguration which can lead to "hard" crashing errors or (possibly gradual) performance problems. Debugging of such errors is an expensive and tedious work, as they frequently manifest in a deployment scenario. For example, a misconfiguration in the user authentication system caused a breakdown of some essential Google services (e.g., Gmail and Drive) for two hours in April 2013, see [2].

A misconfiguration can assume many forms, including mistaken parameter values, inconsistencies between option values, and illegal parameters. According to Yin et al. [12], mistaken parameter values account for 70-85% of all misconfigurations (investigation on data on five significant applications), and 40-50% of them result from invalid values. A significant portion of configuration errors can lead to a system crash. We focus in this work on this type of defects.

Most execution environments save a stack trace once a crashing failure has occurred. Our basic idea is to analyze dependencies between each configuration option and the stack trace to diagnose misconfigurations. If one of the code locations referred by the stack trace has a direct data or control dependency to a configuration option, the latter is quite likely to be the root cause of the failure.

Our approach uses a static analysis technique to identify program statements affected by each configuration option. Let A be as set of such statements (for a specific configuration option). When a user encounters one configuration error, she first needs to extract statement locations from the stack trace. Our approach subsequently identifies statements affecting the stack trace, and records them in a set B. If there is an intersection between sets A and B, the corresponding configuration options are listed in a set of suspects.

Since static analysis involves all possible execution paths, too many configuration options will be treated as suspicions. To address this problem, we adopt filtering technique to narrow down the range of the root causes of a configuration error. After this filtering, we report the set of suspicious configuration options to the user.

Our approach has several advantages compared to the previous solutions to this problem [1], [5], [9], [11], [15]. Firstly, it only needs the configuration options, the stack trace of the failure, and the bytecode of Java program. User does not need to provide any additional information. This is more efficient compared to delta debugging [13], information flow analysis [1] and dynamic slicing [16]. Secondly, differently to the recent approaches [5] and [15] which combine static and dynamic analysis, our approach only uses static analysis. Finally, it does not require any profiles of the application executed with correct configuration like the method in [15].

In summary, this paper makes the following contributions.

- **Technique.** We propose a new lightweight approach to locate the root causes of configuration errors.It is able to diagnose the misconfiguration using only static analysis of code (Section II).
- **Implementation.** We implement the proposed technique and develop a tool called ConfDebugger (Section III).
- **Evaluation.** We evaluate our approach on 8 configuration errors in Java program JChord [4]. The results in Section IV show that it features high precisionin detecting of the root cause of a misconfiguration (Section IV).

The rest of the paper is organized as follows. The approach is described in Section II and its implementation in Section III. The results of the experiments are discussed in Section IV. Related work is presented in Section V and conclusions in Section VI.

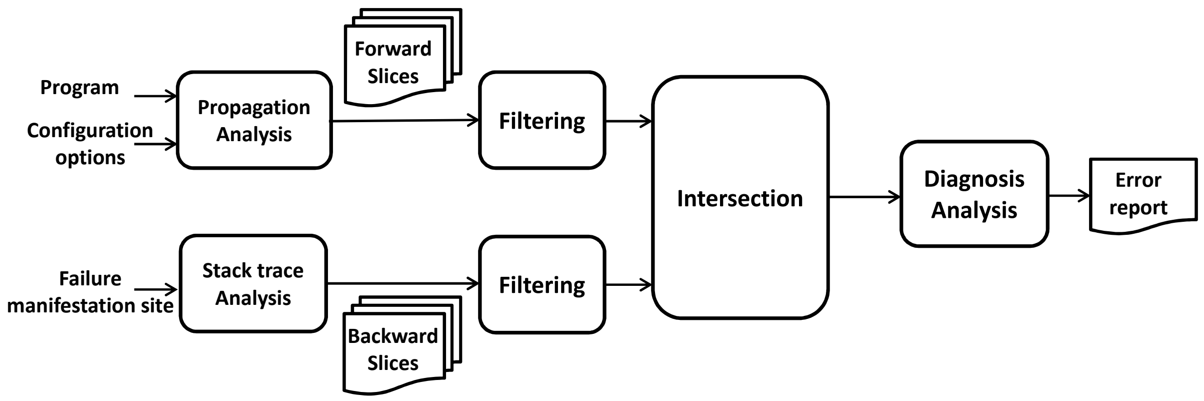

Figure 1. Workflow of our diagnosis technique

II. Description of the Approach

This section describes in detail the proposed approach for diagnosing configuration errors, starting with some definitions.

A *software configuration* is comprised of a set of *configuration options* (also called *configuration parameters*). Each option is a read-only constant variable which modifies certain aspects of program behavior.

A typical way to specify the option values is a *key-value* style configuration. Here the keys are strings, each naming the configuration parameter. The corresponding value (commonly, also a string) specifies the value of this parameter.

For example, in the Chord/JChord framework for Java bytecode analysis (Section IV-A), a file named `chord.properties` holds all configuration settings (in a so-called *Java Properties* format). Each line in this file corresponds to one configuration option. For Chord, one of them specifies the main class of the program to be analysed as `chord.main.class=foo.Main`. Here `chord.main.class` is the key (option name) and `foo.Main` is its value.

This key-value configuration schema is supported by the POSIX system environment, Java Properties and Windows Registry, and used in a range of open source projects. It is also assumed in many related works on configuration debugging, see e.g. [5].

A. Overview

An overview of our technique is given in Algorithm 1 and additionally illustrated in Figure 1. It uses as input a source code and bytecode of a Java program, its particular set of configuration options and a failure stack trace. Firstly, the affected statements of the program are identified by each configuration option by performing a propagation analysis (Section II-B). We call these sets of statements the *forward sets*.

Our technique conducts then failure stack trace analysis to identify the program statements which affect stack trace directly (Section II-C). We call such sets of statements the *backward sets*. After filtering the unnecessary statements from forward and backward sets (Section II-D), we compute the intersection of both sets (Section II-E). Finally, we identify the root cause of the misconfiguration based on the statements in the intersection and then report a list of suspicious configuration options to users (Section II-F).

Algorithm 1 Steps of the proposed approach

Step 1: Compute Forward slices for each of n configuration options:

$$\begin{aligned} FSlice_i &= \text{ForwardSlice}\,(app, conf_i),\ i \in \{1, ..., n\}, \\ FSlice &= \sum_{i=1}^{n} FSlice_i \end{aligned}$$

Step 2: Extract Error Set E_{set} from the stack trace

Step 3: Compute backward slices for each element e of E_{set}:

$$\begin{aligned} BSlice_j &= \text{BackwardSlice}\,(app, e),\ j \in \{1, ..., m\} \\ BSlice &= \sum_{i=1}^{m} BSlice_i \end{aligned}$$

Step 4: Filter $FSlice$ and $BSlice$ using breadth-first search distance algorithm

Step 5: Compute the intersection of forward and backward slices:

$$IS = FSlice \cap BSlice$$

Step 6: Get list of suspects by analyzing the intersection IS

B. Configuration Propagation Analysis

The basic technique for analyzing propagation of configuration option values is *program slicing* [10]. For a given statement s of a program (a *seed statement*), the forward slice of s is a subset of the program statements whose execution may be affected by statement s.

Parts of source code whose execution is potentially affected by a configuration option can be identified by using the forward slices. To this end we first need to identify the *entry point* of a configuration option value, i.e. a part of source code which reads in a particular option value during the execution. This entry point is used as a seed statement for computing a forward slice. To identify the entry point of a configuration option, a simple way is searching the statements which read the value of a configuration option by using a call graph of a program.

The traditional full slicing [3] systematically identifies parts of the program which can affect or are affected by a specific statement. However, the slices created in this way are usually too large.

To address this limitation, we use *thin slicing* [6]. This technique yields the subset of statements which are directly affected by a configuration option. Essentially, it excludes statements involving base pointers and control flow dependencies. This can significantly decrease the size of the slice.

C. Stack Trace Analysis

Java applications (as well as many operating systems) commonly produce a stack trace when a crash happens, not just a simple error message. In Java, the stack trace lists a hierarchy of nested methods called up to the point of failure (including the code location of each call). The top line of the stack trace is the point where an exception is raised.

However, an option misconfiguration can affect *any* of the methods listed by the stack trace. Consequently, we analyze all lines of the stack trace to find statements relevant to the failure in the program. Failure stack trace analysis is comprised of the preprocessing of the stack trace and the stack trace analysis by backward slicing.

1) Preprocessing of the stacktrace: In Java programs, developers typically handle exceptions with a dedicated class. This leads to a phenomenon that stack traces from different failures have the same first line. This line corresponds to an exception and does not help to identify root cause of errors.

In our approach, we check the first line of the stack trace. If this points to an exception-handling statement, we ignore this line. Similarly, we do not consider the bottom line of the stack trace which is the entry point to the main method of the program.

In addition, some of the methods can be called more than once during the execution of an application (see Figure 2). Consequently, if multiple entries of the stack trace point to the same code location, only one of these entries is considered.

Note that the preprocessing of the stack trace is not automated in our current prototype and needs to be completed manually. The problem is that it is very difficult to identify the exception-handling statements (among those pointed by the stack trace). Nevertheless, we will attempt to automatize this step as a part of the future work.

After preprocessing, we map the remaining stack trace entries to code locations (indicated by pairs (*className*, *lineNumber*)). The set of these locations is called the *Error Set*.

```
Exception in thread "main" java.lang.NoClassDefFoundError:
notexist
  ...
  ...
at chord.project...runTask(ClassicProject.java:393)
at chord.project...runTask(ClassicProject.java:390)
at chord.project...runTask(ClassicProject.java:390)
at chord.project...runTask(ClassicProject.java:414)
at chord.analyses...run(DataraceAnalysis.java:90)
at chord.project...runTask(ClassicProject.java:393)
at chord.project...runTask(ClassicProject.java:414)
  ...
  ...
```

Figure 2. A fragment of the stack trace for failure #1 from Table I

2) Stack trace analysis by backward slicing: In this step, we use each of the statements from the *Error Set* as the seed statement to compute the *backward slice*. The backward slice is a subset of statements that may affect the value of any variables at the seed statement [14]. Also in this case we use thin slicing [6] instead of traditional full slicing [3].

Contrary to forward slicing, backward slicing not only considers data dependencies but also control dependencies. Using also control dependencies has two advantages. Firstly, stack trace is a sequence of method calls, which itself is a reflection of control flow when a crash occurs. Considering control dependencies can significantly help to trace the root cause of the error.

Secondly, statements which are indicated by stack trace are method calls. Many of them do not directly involve data dependencies. If the control dependencies are not considered, the backward slice would only contain the seed statement.

D. Filtering

Thin slicing dramatically reduces the slice size compared to the traditional full slicing. However, the size of slices in the configuration propagation and the failure stack trace analysis are still large, especially when the control dependencies are considered in the failure stack trace analysis. In face of this fact we further prune slices and remove statements which are less relevant to the seed statement.

In thin slicing, statements "closer" to the seed statement are more likely to be relevant to seed's behavior. Consequently, for a statement s we measure its breadth-first search distance from the seed statement in the dependence graph [6]. Our filtering works then as follows: if this distance exceeds a certain threshold, it is excluded from the slice.

E. Failure-inducing Chop

The basic idea used in our approach is that a configuration option is more likely to be the root cause of the misconfiguration if there exists an intersection between its forward slice (Section II-B) and the backward slice of the stack trace (Section II-C). We call this intersection as the *failure-inducing Chop* (*FChop*).

In Step 5 of Algorithm 1 we compute the statements in the intersection *FChop* of the forward slice and the backward slice.

F. Configuration Diagnosis Analysis

In this section, we analyze the statements in the *FChop* to identify the root cause of the error.

Three cases are taken into account. Firstly, statements in the *FChop* only correspond to one configuration option. We call this case a *single suspect*. Secondly, statements in the *FChop* correspond to multiple configuration options, called *multiple suspects*. Lastly, *FChop* is empty. We call this case as *no suspect*. In the following we explain each of these three cases in detail.

1) Single Suspect: This situation is the optimal outcome for our technique. The configuration option in the *FChop* is reported to the user as the root cause of the misconfiguration.

2) Multiple Suspects: If the statements in the *FChop* link to more than one configuration option, we use filtering to narrow down the range of the suspicious configuration options by adopting the following strategy.

In configuration propagation analysis, if the forward slice of a configuration option contains any statement in the *Error Set,* it means that there exist dependencies between the configuration option and the error. We say in this case that the configuration option can *reach* the error.

On the other hand, if an entry statement of a configuration option is contained in the backward slice of the stack trace, it means that the error can reach the configuration option in the stack trace analysis.

If a configuration option and the stack trace can reach each other, this configuration option has a higher probability to be the root cause of the error than other configuration options. We report these configuration options as the root causes of the misconfiguration.

3) No Suspect: For the situation that the *FChop* is empty, we remedy it by relaxing the conditions of making the failure-inducing chop. Firstly, the threshold on breadth-first search distance (II-D) is increased until the *FChop* is not empty. If the *FChop* is still empty, we use an approximate failure-inducing chop (*AFChop*) as follows.

During computing intersection of forward and backward slices, if the source line numbers of two statements in the same class file are close to each other, not just exactly equal, we consider the two statements as the same statement and put them into the *AFChop*. The mathematical description of the *AFChop* is as follows:

Defining C as class name and L as source code line number, statement $S_f(C_f, L_f)$ is in the forward slice of configuration options and statement $S_b(C_b, L_b)$ is in the backward slice of the stack trace. Then $S_f$and S_b belong to the *AFChop*, if the following three conditions are satisfied:

1) $C_f = C_b$,
2) $L_b - L_f \leq \varepsilon$, where ε is a predefined threshold
3) S_f and S_b are in the same method.

We increase ε until *AFChop* is not empty. Then, the corresponding configuration options in *AFChop* will be reported to the user.

G. Discussion

Why are forward and backward slicing combined? The forward slice of an entry point of an erroneous configuration option should reach the failure stack trace. If this is the case, this configuration option can be assumed to be the root cause of this defect (this idea is used in paper [5]). Similarly, an entry point of an erroneous configuration option which is reached by the backward slice of the failure stack trace should be the root cause of the error. So why does it still make sense to combine both techniques?

The answer is that by using both types of slicing we can increase the specificity of our approach. Static analysis involves all possible executable paths (instead of only executed paths as in dynamic analysis). Consequently, entry points of multiple configuration options can reach the failure stack trace in the forward slice analysis. On the other hand, the backward slice of the failure stack trace can also reach the entry points of multiple configuration options. Therefore, we combine both types of slices to narrow down the range of the suspicious configuration options.

How can *FChop* be empty? Due to simplifying assumptions and implementation constraints, the static analysis tool used in this work does not give all the statements affected by a seed statement. Consequently, it can ignore statements which are logically related to the seed statement. This leads to an empty intersection between the forward slice of configuration options and backward slice of the failure stack trace. We will further discuss it in our evaluation (Section IV).

III. Implementation

We implemented a tool, called *ConfDebugger*, on top of the WALA [8]. WALA is a static analyzer tool developed by IBM. It analyzes Java bytecode and locates the entry statements based on the list of configuration options. Then it computes the subsets of the statements affected by each configuration option and the statements affecting the failure stack trace. After that, our prototype computes and assigns the breadth-first search distance to the seed statement for every statement, filters statements, makes the *FChop* (*AFChop*), and performs the analysis on the *FChop* (*AFChop*).

Our prototype does not analyze the standard JDK library and all libraries which subject program depends on. We believe it makes sense, since the configuration settings of an application almost never affect the behaviors of its dependent libraries.

To achieve scalability, we do not consider data dependencies involving heap. Considering such dependencies would incur a very significant additional memory overhead by the WALA tool.

Currently we cannot exactly say whether using data dependencies involving heap would improve the results or not. However, at least in case of the error #6 (Section II-F2) an improvement is likely. If we would be able to use heap-related dependencies, then he root cause of this error could be possibly determined more easily - using FChop instead of AFChop.

Table I
THE 8 CRASHING CONFIGURATION ERRORS USED IN THE EVALUATION

Error ID	Crashing errors in 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 name
5	Printing nonexistent relations
6	Disassembling nonexistent classes
7	Invalid reflection kind
8	Wrong classpath

IV. EVALUATION

In this section, we evaluate the effectiveness of our technique by the following aspects:

- the precision of the analysis results
- the time effort of error diagnosis
- comparison with previous configuration error diagnosis techniques.

Besides, we evaluate the impact of the filtering (by breadth-first search distance) on the precision of analysis.

A. Experimental Environment and Configuration Errors

We have selected JChord (version 2.1) [4] as our application for experimenting. JChord is a program analysis platform that enables users to design, implement, and evaluate static and dynamic program analysis for Java bytecode. JChord is used as a subject program in multiple papers on configuration errors diagnosis. By selecting this application we are able to easily compare analysis results against those stated in the related work.

We considered 9 crashing configuration errors which were previously used to evaluate the ConfAnalyzer [5] and the ConfDiagnoser [15] tools. Before diagnosing, we attempted to reproduce all 9 of these crashing errors. However, one of them could not be reproduced because of different versions of JChord. We thus use the remaining 8 errors to evaluate our technique (see Table I).

Our experiments were conducted on a dual core 2.00 GHz Intel PC with 4 GB physical memory, running Windows 7.

B. Results

1) Precision of diagnosing configuration errors: As shown in Table II, ConfDebugger is highly effective in pinpointing the root cause of software configuration errors. It successfully diagnoses all configuration errors except one. The success ratio is 87.5%. For 5 of them, it exactly gives the root cause without any false positives. Here a false positive is a correct configuration option which is reported as a suspicious one. The remaining 2 errors have 1 false positive each. The average number of false positives was 0.5.

For errors #3, #5, #7, there is only one statement in the *FChop*. It was the entry statement of the configuration option which is responsible for the error. For errors #1, #2, there are 25 statements which correspond to 13 configuration options in the *FChop*. After analysis (see Section II-F), two configuration options remain in the output list of suspicious configuration

```
71 String[] printClasses =
           Utils.toArray(Config.printClasses);
72 if (printClasses.length > 0) {
73 for (String className : printClasses)
74 program.printClass(className);
75 }
```

Figure 3. Exception of the JCord related to error #6

Table III
EXECUTION TIME OF VARIOUS TASKS OF CONFDEBUGGER (SECONDS)

Error ID	Forward Slicing	Backward Slicing	Analysis
1	21	35	2
2	21	35	2
3	21	18	< 1
4	21	21	< 1
5	21	14	< 1
6	21	12	2
7	21	13	< 1
8	21	35	< 1

options. The *FChop* of error #4 contains 4 statements corresponding to 3 configuration options. Using multiple suspects strategy (see Section II-F2), our approach identifies one of them as the root cause of the error.

For error #6, the *FChop* is empty. The reason is that the value of this configuration option directly flows into a container after it enters the program (see Figure 3). WALA analyses that the configuration option *Config.printClasses* does not affect the lines 72, 73 and 74. It shows that it is not effectively propagated. On the other hand, the backward slice of the stack trace can not reach line 71. Consequently, the *FChop* can not diagnose this error.

After relaxing the conditions and using *AFChop* (see Section II-F3), lines 71 and 72 are only included in *AFChop* when the threshold ε is set to 1. After this adjustment, the configuration option *chord.print.classes* is reported by our approach as the root cause of the configuration error.

ConfDebugger cannot diagnose error #8. The reason is that this error involves the system calls and does not produce any information related to the root cause of the misconfiguration (*chord.class.path*) in the stack trace. The stack trace generated for this error is the same as for error #1. The current implementation of ConfDebugger cannot handle this type of errors. This could be a good point for further extensions of our approach.

2) Performance Aspects: We evaluated the performance of ConfDebugger in terms of time effort for diagnosing a configuration error. As shown in Table III, the time effort essentially is caused by three tasks: forward slicing, backward slicing and analysis. Forward slicing uses 21 seconds, which is one-time effort per program (as slicing results can be cached and then used for all of configuration options of the program). The time of backward slicing depends on the size of preprocessed stack trace. Analysis operation uses less than 2 seconds.

Overall, the total time is less than 1 minute. This is fast enough for most usage scenarios. Here, we ignore the time of

Table II
EXPERIMENTAL RESULTS OF DIAGNOSING SOFTWARE CONFIGURATION ERRORS WITH DIFFERENT APPROACHES. DATA IN THE COLUMNS "CONFANALYZER" AND "CONFDIAGNOSER" IS TAKEN FROM [5] AND [15], RESPECTIVELY.

Error ID	**Erroneous Configuration Option**	**ConfDebugger**		**ConfAnalyzer** [5]		**ConfDiagnoser** [15]		**No Filtering**	
		# False positives	Success	# False positives	Success	# False positives	Success	# False positives	Success
1	chord.main.class	1	Y	1	Y	0	Y	1	Y
2	chord.main.class	1	Y	0	Y	0	Y	1	Y
3	chord.run.analyses	0	Y	3	Y	16	Y	21	Y
4	chord.ctxt.kind	0	Y	1	Y	0	Y	4	Y
5	chord.print.rels	0	Y	0	Y	14	Y	8	Y
6	chord.print.classes	0	Y	0	Y	15	Y	3	N
7	chord.reflect.kind	0	Y	4	Y	0	Y	4	Y
8	chord.class.path	2	N	2	N	7	Y	2	N
Ave. # of false positives | Success ratio in %		0.5	87.5%	1.7	88.9%	5.7	100%	5.5	75%

importing statements into database.

3) Comparison with related techniques: We choose ConfAnalyzer and ConfDiagnoser to make comparison with our technique. They are two of the most recent and precise techniques described in the literature of the misconfiguration diagnosis.

ConfAnalyzer labels all the configurations, tracks the flow of the labels by static data flow analysis, and treats a configuration option as the root cause if its value flows into the crashing point. Like our technique, ConfAnalyzer focuses on crashing errors.

ConfDiagnoser uses static analysis to identify predicates affected by a configuration option, and records the behavior of them. If the behavior of the predicates corresponding to a configuration option is very different from behavior of predicates in the program with correct configuration, then ConfDiagnoser identifies this configuration option as the root cause of the error.

In comparison with the result of ConfAnalyzer shown in Table II (Column "ConfAnalyzer"), ConfDebugger produces a better result for 3 of 8 errors, the same results for 4 errors, and worse result for only 1 remaining error. In terms of the averaged number of false positives, ConfDebugger's score is better than ConfAnalyzer. Both of ConfDebugger and ConfAnalyzer cannot diagnose the error #8, because the value of the root cause configuration option flows into system calls.

In comparison with ConfDiagnoser, the result in Table II shows that ConfDiagnoser successfully pinpoints the root cause for 4 errors without false positives. For the remaining errors, it gives too many false positives. The average false positives is much higher than ConfDebugger.

A significant advantage of ConfDiagnoser is that it can diagnose error #8. The reason is that ConfDiagnoser observes behaviors of predicates affected by configuration options to identify the root cause of the error instead of tracking the configuration options values. However, it needs to build a database for correct profiles.

C. The Effectiveness of the Breadth-First Search Distance Filter

In ConfDebugger, we filter slices by using the breadth-first search distance (Section II-D). In the experiment, we set 1 as the threshold. The statements whose breadth-first search distance to the seed statement is larger than 1 are excluded from the slice.

As shown in Table II (Column "No Filtering"), ConfDebugger achieves substantially less accurate results without filtering. The primary reason is that the slice without filtering contains too many irrelevant statements. They are indirectly affected by a configuration option but not helpful in diagnosing configuration errors. When computing the *FChop*, a lot of configuration options are included in results, which leads to a low accuracy. We can see that it fails in diagnosing error #6. The reason is that the *FChop* contains some irrelevant statements. The *AFChop* is not triggered. We conclude that the breadth-first search distance filter is able to significantly improve the precision of our technique.

D. Experimental Discussion

Summary of results. The following conclusions can be drawn from the above evaluation.

- ConfDebugger is a highly effective tool for diagnosing configuration errors. It can produce more precise diagnosis result for crashing misconfiguration errors than existing tools ConfAnalyzer and ConfDiagnoser.
- ConfDebugger shows a good performance in terms of time effort for diagnosing a configuration error.
- The heuristic of the breadth-first search distance filter (Section II-D) can significantly increase the accuracy of diagnosis results.

Limitations. Several limitations exist in the experiments. Firstly, we only focus on Java applications with key-value style configuration options. Secondly, the errors we used in the experiments involve one misconfiguration option. Thirdly, our technique just works on the crashing errors with a generated stack trace. Lastly, we assume the application code is correct. In fact, a user does not know whether the problem is related to the application code or configuration when he gets an error. Our technique can not indicate whether an error is raised from a misconfiguration.

Threads to Validity. There are some threads to validity in our evaluation. Firstly, although the errors used in the experiments have been used in multiple papers before, it is not representative. We did not evaluate our technique on other programs. Secondly, we use *AFChop* to cope with the situation that *FChop* is empty. Its precision significantly depends on the structure of a program, or the programming habits of the developers.

V. Related Work

This section summarizes the closely-related work on software configuration error diagnosis. We generally group the existing techniques into two areas: program analysis approaches and non-program analysis approaches.

Program analysis. We are aware of three instances of prior work using program analysis for configuration diagnosis: ConfDiagnoser [15], ConfAnalyzer [5] and ConfAid [1].

ConfDiagnoser [15] uses a combination of static and dynamic analysis technique to record run-time behavior of predicates affected by configuration options in the execution profile. When predicates affected by a configuration option behavior differently compared to correct profiles, ConfDiagnoser considers this configuration option as a suspicious root cause of the error.

In ConAnalayzer [5], a map between configuration options and source code is built. Then using this map and static analysis data flow, the faulty configuration options are detected.

In ConfAid [1], at first the dynamic dependencies are recorded using code instrumenting. After that using the obtained dependencies, the faulty behavior connects to the configuration parameters.

Our approach belongs to this direction of researches in misconfiguration diagnosis which attempts to improve the result of predecessor approaches.

Non-program analysis. This direction attempts to find the root cause of misconfiguration without any source code analysis. PeerPressure [9] uses statistical techniques to locate the misconfiguration with comparing a large number of healthy machines with a non-healthy one. The different value of a registry entry in a specific machine in comparison with usual value adopted by other machines can evaluate as a flag for a misconfiguration. OS-level speculative execution is used by AutoBash [7] which examines different possible configurations and chooses the best one to fix the misconfiguration. In [11] they try to find the time when the system changes from correct to faulty state with checking the behavior of the system using a pre-defined testing oracle. Our technique differs from each of these approaches. It only uses static analysis and just needs configuration options, stack trace of the failure and the bytecode of Java program without any additional information. In addition, it does not need any profiles of the application with correct configuration.

VI. Conclusion and Future Work

This paper presents a static analysis technique for diagnosing configuration errors. It can identify the root cause by determining whether the direct data or control dependence exists between a configuration option and stack trace of an error.

We evaluated the technique on 8 configuration errors. Result showed that it can successfully diagnose the root cause for 7 errors. The average number of false positives was 0.5. In comparison with recent techniques, our technique had a better performance in terms of average number of false positives.

As a part of the future work we will evaluate how it generalizes by applying it to other application programs than JChord. We will also target some deficiencies of the current work. For example, when a configuration option value flows into containers, our technique cannot locate the statements logically related to the configuration option. Another area for future work is to explore diagnosing inconsistencies in configuration settings.

Acknowledgments

We thank Felix Langner and Lutz Büch in PVS group of Heidelberg University for their useful suggestions. We appreciate time and effort of the anonymous reviewers to appraise this work.

References

[1] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.
[2] Report. http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/en/us/appsstatus/ir/ej73a82sddnv7fb.pdf.
[3] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.
[4] JChord. http://pag.gatech.edu/chord/.
[5] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *ASE*, 2011.
[6] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.
[7] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: improving configuration management with operating system causality analysis. In *SOSP*, 2007.
[8] WALA. http://sourceforge.net/projects/wala/.
[9] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.
[10] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, 1984.
[11] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI*, 2004.
[12] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.
[13] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.
[14] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
[15] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *ICSE*, 2013.
[16] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, 2003.