Practical and Accurate Pinpointing of Configuration Errors using Static Analysis

Zhen Dong
Institute of Computer Science
Heidelberg University, Germany
zhen.dong@informatik.uni-heidelberg.de

Artur Andrzejak
Institute of Computer Science
Heidelberg University, Germany
artur.andrzejak@informatik.uni-heidelberg.de

Kun Shao
School of Computer and Information
Hefei University of Technology, China
shaokun@hfut.edu.cn

***Abstract*—Software misconfigurations are responsible for a substantial part of today's system failures, causing about one-quarter of all customer-reported issues. Identifying their root causes can be costly in terms of time and human resources. We present an approach to automatically pinpoint such defects without error reproduction. It uses static analysis to infer the correlation degree between each configuration option and program sites affected by an exception. The only run-time information required by our approach is the stack trace of a failure. This is an essential advantage compared to existing approaches which require to reproduce errors or to provide testing oracles. We evaluate our approach on 29 errors from 4 configurable software programs, namely JChord, Randoop, Hadoop, and Hbase. Our approach can successfully diagnose 27 out of 29 errors. For 20 errors, the failure-inducing configuration option is ranked first.**

Keywords*—*Configuration errors, Static analysis, Automated debugging

I. Introduction

Software configuration errors are one of the major causes of today's system failures [7], [16], [17], [32], [18]. The investigation in [33] shows that around 27% of the issues in one company's costumer-support database are labeled as configuration-related. Recently misconfiguration-induced outages have been reported from major IT companies, including Microsoft, Amazon and Facebook [25], [26], [12]. Moreover, end-users also suffer from various configuration errors of software.

According to a recent study [33], parameter-related misconfigurations account for 70%-85% of all users' configuration errors. A parameter-related misconfiguration is considered that one or more configuration parameters (or configuration options) are set mistakenly by users. The mistaken settings lead to abnormal behavior of a system. Our work addresses one type of parameter-related misconfigurations, a crashing error caused by the incorrect value of a single configuration option.

Diagnosing such errors is both time-consuming and tedious. Most applications have no dedicated repository of configuration issues. Seeking help from the software developers usually takes long time to get responses.

To address this dilemma, many groups [27], [1], [5], [14], [19], [24], [35], [2], [37], [23] have devoted themselves to develop tools for automated debugging of configuration errors. The current generation of these tools and approaches require extensive data from the failing program run. For instance, ConfDiagnoser [37] adopts dynamic analysis to record behavior of the program with mistaken configuration. ConfAid [2] employs dynamic analysis to monitor data and control dependence during the failing program execution. All these tools assume that errors can be reproduced.

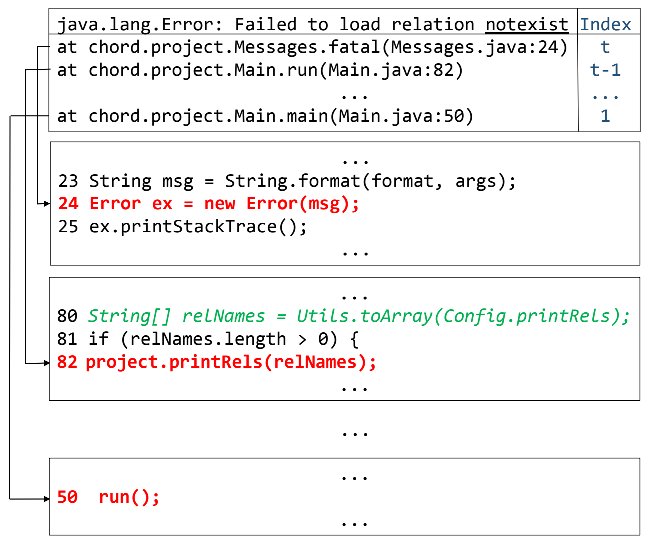
In the real world, reproducing even classical (i.e. code) errors is difficult. Errors often occur during production runs. Users prefer not to report all essential information to reproduce an error because of privacy and economic concerns. Research [38] has shown that there is a strong mismatch between what developers need to reproduce and fix a bug and what users tend to provide. Another study [3] 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. Additionally, for distributed systems, the root cause of a misconfiguration can be an incorrect setting at remote nodes, which creates additional cost of error reproduction.

Providing values of configuration setting, failure-inducing program input, and environment information can conflict with confidentiality concerns of a user. Error reproduction in data-sensitive scenarios is likely to be performed only by the user. This precludes the involvement of experienced third parties such as application developers.

Aiming at the issue above, we propose an approach capable of diagnosing misconfigurations with only minimal runtime information collected during the failing run. Specifically, in the preprocessing phase our approach needs as input the program source code (or compiled, not obscured Java bytecode) and a list (but not values) of available configuration options. All this data is independent of particular deployment scenario and frequently publicly available, e.g. in case of open source software. For the diagnosis step, users need to provide the stack trace of an error, and no other execution-related data.

The approach is inspired by the way how developers



```
java.lang.Error: Failed to load relation notexist       Index
at chord.project.Messages.fatal(Messages.java:24)        t
at chord.project.Main.run(Main.java:82)                  t-1
                        ...                              ...
at chord.project.Main.main(Main.java:50)                 1
```

```
                        ...
23 String msg = String.format(format, args);
24 Error ex = new Error(msg);
25 ex.printStackTrace();
                        ...
```

```
                        ...
80 String[] relNames = Utils.toArray(Config.printRels);
81 if (relNames.length > 0) {
82 project.printRels(relNames);
                        ...
```

```
                        ...
```

```
                        ...
50  run();
                        ...
```

Figure 1. Example showing how developers diagnose a configuration error based on the stack trace. The statements in bold (red) are program points referenced by the stack trace entries. The statement in italic (green) is a read point of a configuration option.

typically debug erroneous configuration settings using a stack trace. As shown in Figure 1 the first step is usually to check the top frame of the stack trace. 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. In Figure 1, the root cause candidate can be detected from the analysis of the second frame of the stack trace.

Based on our previous work, called *ConfDebugger* [6], this paper proposes a systematic approach to compute the strengths of potential "links" between each configuration option and program sites where the error manifests. On this basis we are able to rank options by their likelihood to be the root cause of a failure. The contributions of this work are following ones.

- We propose an automated approach for diagnosing configuration errors which relies only on static analysis. The approach does not require reproducing errors. 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. There is no need for program re-execution, code instrumentation or modifications of the current configuration settings.
- We evaluate the accuracy of our approach and its implementation *ConfDoctor* 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.

In Section II we detail our approach. Section III discusses the implementation. In Section IV we evaluate the accuracy of the overall approach, and compare the results against our previous work, ConfDebugger. Section V discusses related work, and Section VI states our conclusions.

II. Approach Description

Our approach considers the source code of a program as a set of *statements* S_1, $S_2, \ldots$. 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 [19].

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, $\ldots$), letters P, R, S, Q denote statements, M, N are methods, and X, Y, Z are sets.

A. Overview

Our approach implies a following diagnosis workflow. For a targeted application we first perform a one-time configuration propagation analysis (Section II-B) to identify statements possibly affected by values of each configuration option. Given a crashing error and its error stack trace (Figure 1) we conduct a backward slicing analysis (Section II-C) to identify statements which impact program points referenced by this trace. As next an intersection of both sets of statements is computed (called *chopping*, Section II-D). We use this result to correlate each configuration option with a given error stack trace (Section II-E). Finally, a list of configuration options ranked by the correlation degree is reported to users.

B. Configuration propagation analysis

The propagation analysis consists of two steps.

Searching option read points. We assume that configuration options of a software program are published. Users are allowed to acquire the list of configuration options. 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. To identify all statements affected by a configuration option we use a static technique called *forward slicing* [30]. For a *seed* statement S, it attempts to identify the set of all statements (called *forward slice* FS(S)) affected by the execution of S.

We deploy a variant of forward slicing which considers data dependence without control dependence (Section III).

The reason is that considering control dependence includes in slices 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* MFS(c_i) as the union of all forward slices over all option read points of c_i:

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

C. Stack trace analysis

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 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 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* [30], 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. Details are omitted due to space constraints.

D. Chopping

The core idea of our approach 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 2, 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 (per option) and multiple FEPs,

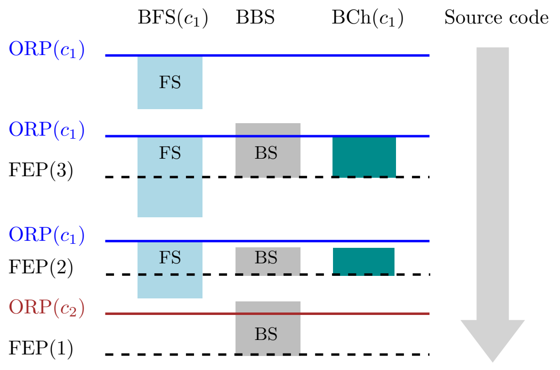


Figure 2. An example illustrates how the option read points ORPs of configuration options c_1 and c_2 and frame execution points FEPs of an exception give rise to the merged forward slice MFS(c_1) of c_1, the merged backward slice MBS, and the merged chop MCh(c_1)

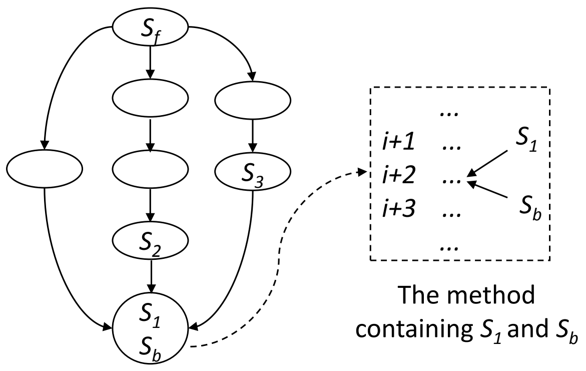


Figure 3. A fragment of a call graph with call paths from the method containing S_f to the method containing S_b

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:

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

E. Correlation degrees

This section describes two variants of metrics used for ranking of configuration options based on the results of the propagation analysis and stack trace analysis. We first introduce some definitions.

Method distance. 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 CG between a method containing S_p and a method containing S_q. Obviously $d_{\text{meth}}(S_p, S_q) = 1$ 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 3 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 3 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 FS(S_f) $\bigcap$ 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 FS(S_f) $\cap$ 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 3, 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 FS(S_f) $\cap$ 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 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.

1) 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* $Cor_{fw}(c_i)$:

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

We set $Cor_{fw}(c_i) = 0$ if there is no pair $S_f \in X$, $S_b \in Y$ with FS(S_f) $\cap$ 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

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

Again we set $Cor_{bw}(c_i) = 0$ if there is no pair $S_f \in X$, $S_b \in Y$ with FS(S_f) $\cap$ 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 compute statistics 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 $(S_f^{fw}, S_b^{fw}) = \text{argmax}_{S_f, S_b} D_{fw}(S_f, S_b)$ and let $(S_f^{bw}, S_b^{bw}) = \text{argmax}_{S_f, S_b} D_{bw}(S_f, S_b)$. Then the *minimal ORP to FEP distance* $d_{min}(c_i)$ is a smaller one of the method distances $d_{\text{meth}}(S_f^{fw}, S_b^{fw})$ and $d_{\text{meth}}(S_f^{bw}, S_b^{bw})$. The index of the stack trace entry corresponding to S_b^{fw} (or to S_b^{bw} if $d_{\text{meth}}(S_f^{bw}, S_b^{bw})$ is used) is called the *key frame* for c_i.

2) 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 [21] 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.

To consider the impact of stack frame ordering we introduce an *stack order factor* f:

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

where f is a weight function for each stack frame and j is the index of a stack frame (see Figure 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:

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

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 $\text{FS}(S_f) \cap \text{BS}(S_b) \neq \emptyset$).

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

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

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*:

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

F. Ranking configuration options

As noted in Section II-A, after the error stack trace is available, we compute the correlation degrees for all configuration options as described in Section II-E. 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.

III. IMPLEMENTATION

We implemented a Java-based prototype, called ConfDoctor, on top of the WALA library [28]. WALA is a static analyzer tool developed by IBM. We use it to compute forward and backward slices. In particular, in a default approach we use for both slicing types NO_BASE_NO_HEAP_NO_EXCEPTIONS as the value of the DataDependence parameter. 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 achievehigher precision, we use here a more expensive algorithm, the Control Flow Analysis 0-CFA [8]. 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.

Program (version)	Lines of Code	#Options
JChord (2.1)	23391	79
Randoop (1.3.2)	18587	57
Hadoop (0.20.2)	103649	141
HBase (0.92.2)	187433	91

Table I. BENCHMARK APPLICATIONS. COLUMN “LINES OF CODE” IS THE NUMBER OF LINES OF CODE AS COUNTED BY CLOC [4]. FOR HADOOP AND HBASE, IT IS THE NUMBER OF LINES OF JAVA SOURCE CODE. COLUMN “#OPTIONS” IS THE NUMBER OF AVAILABLE CONFIGURATION OPTIONS.

IV. 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 [6].

A. Experimental setup

1) Subject applications: We evaluated ConfDoctor on four Java programs shown in Table I. JChord [11] is a program analysis platform for Java byte code initiated by Mayur Naik and Alex Aiken at Stanford University [19]. Randoop [20] is an automatic unit test generator for Java maintained by a product group at Microsoft. Apache Hadoop [9] comprises a distributed file system, a MapReduce implementation, and a job scheduling/cluster management framework. Apache Hbase [10] is a distributed, scalable, big data store which runs on top of Apache Hadoop’s file system.

2) Configuration errors: We collected 29 configuration errors[1] listed in Table II. We evaluated all the configuration errors we found; we did not remove errors on which ConfDoctor does not work well.

The errors for JChord are taken from [19], and also used in [37]. 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 [13]. 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 [22]. 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

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

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

Table II. CONFIGURATION ERRORS USED IN OUR EVALUATION.

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 [22]. 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.

B. 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 MCh(c_i) is not empty (or equivalently, by definitions in Section II-E, if $Cor^{st}(c_i) > 0$).

Column Cor^{st} in Figure 4 shows the main result of our approach. Notation R/S means that the defective option has rank R in a list with S suspects. 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.

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.

C. Comparison of accuracy of Cor versus 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 II-E1).

The diagnosis results are shown in column Cor of Figure 4. 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 model is based only on the method distance, it is informative to consider the minimal ORP to FEP distance d_{min} defined in Section II-E1. As shown in Figure 4, for all errors of JChord and Randdop the value of d_{min} is 1 except for error #8. As explained in Section IV-B, ConfDoctor cannot successfully diagnose error #8. For Hadoop and HBase, the value of d_{min} is larger, especially for error #27.

The primary reason for these values are 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. A configuration option value might be passed along many methods from being read to being used. The method reading the incorrect value of the configuration option might not appear in the stack traces when an error occurs. For such cases, the value of d_{min} is larger.

	Id	Rank of the root cause		Statistics for rank 1		Variants of dependency analysis			Conf-Debugger
		Cor^{st}	Cor	d_{min}	Key frame	(Ctr, Ctr)	(Ctr, NCtr)	(NCtr, NCtr)	
	1	2/47	2/47	1	18/19	4/64	2/57	2/8	2/2
J	2	1/53	2/53	1	18/19	1/66	1/57	1/8	2/2
C	3	1/45	1/45	1	3/6	5/65	9/19	2/6	1/1
h	4	1/57	2/57	1	6/8	1/69	1/26	1/11	1/1
o	5	1/42	1/42	1	2/4	1/65	2/19	2/6	1/1
r	6	1/37	1/37	1	3/4	2/65	2/8	1/1	1/1
d	7	1/48	2/48	1	3/4	3/69	4/6	4/6	1/1
	8	* 22/47	30/47	3	16/19	28/64	28/57	N	N
	Average	3.8/47	5.1/47	1.3	8.6/10.4	5.6/65.9	6.1/31.1	6.6/15.6	6.1/13.2
	9	1/37	1/37	1	4/6	3/53	2/11	1/3	2/2
R	10	1/35	13/35	1	4/4	1/54	1/8	N	N
a	11	1/47	2/47	1	7/8	3/55	1/20	1/5	2/3
n	12	1/39	1/39	1	3/5	4/54	1/7	1/3	1/1
d	13	1/41	1/41	1	3/11	4/54	1/12	1/3	1/1
o	14	1/41	1/41	1	3/13	4/54	1/13	1/5	1/1
o	15	4/43	2/43	1	4/6	17/53	4/18	4/7	2/4
p	16	1/38	1/38	1	3/5	1/54	1/2	1/1	1/1
	Average	1.4/40.1	2.8/40.1	1.0	3.9/7.2	4.6/53.9	1.5/11.4	4.9 /10.8	4.9/8.8
	17	1/7	1/7	1	6/8	1/32	1/32	1/2	1/1
H	18	1/11	1/11	1	3/8	1/29	1/3	1/1	1/1
a	19	2/7	2/7	2	4/9	9/30	9/21	2/3	N
d	20	1/18	2/18	2	6/9	1/36	1/31	1/1	N
o	21	2/16	2/16	2	6/8	2/38	2/2	2/2	N
o	22	1/11	1/11	1	5/7	1/34	1/29	1/5	1/1
p	23	3/6	3/6	2	4/9	16/31	16/20	2/5	N
	24	1/11	1/11	1	5/7	1/34	1/29	1/5	1/1
	Average	1.5/10.9	1.6/10.9	1.5	4.9/8.1	4.0/33	4.0/20.9	1.4/3	36.1/36.1
	25	1/17	1/17	1	4/4	1/33	1/1	1/1	1/17
H	26	1/17	1/17	1	4/4	1/33	1/1	1/1	1/17
B	27	3/20	8/20	8	9/9	3/33	N	N	16/20
a	28	N	N	-	-	15/32	N	N	N
s	29	3/5	3/5	1	3/5	3/32	N	N	3/5
e	Average	10.8/30	11.8/30	2.2	4/4.4	4.6/32.6	28/55	28/55	13.4/30

Figure 4. Experimental results. 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 the results obtained by the correlation degree with stack order (main output of ConfDoctor). Column Cor shows the ranking by the simple correlation degree. Both metrics are defined in Section II-E. 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 (Table I), assuming that a user would need to examine on average half of options to find the root cause.
Columns under "Statistics for rank 1" show the minimal ORP to FEP distance $d_{min}(c_i)$ and the key frame (Section II-E1) 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 "Variants of dependency analysis" shows the ranks of root causes obtained by Cor^{st} produced by variants of the dependence analysis types (see text). It uses analogous notation as columns for Cor^{st} and for Cor.
Finally, column "ConfDebugger" reports results for our previous work ConfDebugger [6] using an identical notation as for Cor^{st} and Cor.

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 Figure 4 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 drops 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 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 II-E1 is also shown in Figure 4. 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 "upper" half of the error stack trace, i.e. closer to the method where an exception is thrown than to the "main"-method.

D. 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 III 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 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 Figure 4. In terms of average rankings, 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. 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. Overall, the comparison indicates ConfDoctor achieves more accurate results when using the dependence type (*NCtr*, *Ctr*).

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. For instance, 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.

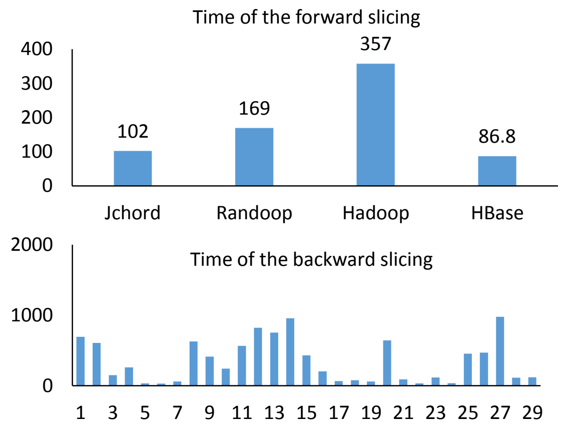


Figure 5. Time of the forward slicing on 4 applications and the backward slicing for each error (*seconds*)

E. Comparison with our previous work

ConfDebugger [6] 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 II-E to compute the dependency between one configuration option and an error.

As shown in Figure 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 II-E) . 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.

F. 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 of the forward slicing on 4 applications and the backward slicing for each error is shown in Figure 5.

The forward slicing does not consider control dependence and takes 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.

G. Discussion

1) 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 deterministic 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. Third, some misconfigurations are caused by the incorrect setting of a combination of multiple configuration options, our approach cannot pinpoint the number of the root cause configuration options. Finally, our implementation and experiments are restricted to Java. It cannot cross components written by different languages.

2) Threats to validity: There are two threats to validity in our evaluation. First, configuration errors for JChord and Randoop are created by using ConfErr [13] 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. Our errors might be not representative. Second, subject application programs may not be representative either, 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.

V. Related Work

Software configuration error diagnosis is recognized as an important research problem and was investigated by many groups from academia and industry. We classify the existing works into two broad areas.

Program analysis. ConfDiagnoser [37] uses dynamic analysis technique to record run-time behavior of predicates affected by configuration options. When these predicates behave differently from the correct profiles, ConfDiagnoser considers this option as a suspect. ConfAnalyzer [19] 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. ConfAid [2] applies dynamic information flow analysis techniques to track tokens from specified "configuration sources" and analyze dependencies between the tokens and the error symptoms, pinpointing which tokens are root causes. SPEX [32] analyze source code to infer configuration option constraints and use these constraints to diagnose software misconfigurations. Sherlog [34] uses static analysis to infer the execution path and state of the program in the run time to diagnose failures.

Our approach falls into this broad category (i.e. program analysis). There are several differences to the approaches above. First, many of them adopt the dynamic analysis technique. ConfDoctor employs only the static analysis technique and does not need users to reproduce errors or execute the instrumented program. Sherlog, and SPEX are similar to our approach. However, Sherlog requires run-time log to infer the execution path. ConfDoctor requires only a stack trace of error. A stack trace is usually smaller in size and easier to record. SPEX has a different objective, namely helping developers improve the configuration design.

Second, ConfDoctor is fully automated, which is targeted at end-users. After initial preprocessing of the source code, a user needs to provide a stack trace of error as input and receives an option ranking within minutes. It is not necessary to re-execute the program nor to instrument it. While Sherlog also refrains from dynamic analysis, it is targeted at product support engineers and not end-users.

Non-program analysis. One category of well-known tools [31][23] are the replay-based diagnosis techniques. They treat the system as a black box to automatically run the system with possible configurations values without damaging the rest of the system until fixing the misconfiguration. This class of techniques relies on having a working configuration. Otherwise, it can not be applied. Besides, they require users with more domain knowledge.

Another family of tools mine a large amount of configuration data from different instances to infer rules about options, and use these rules to identify software misconfigurations. EnCore [36] and CODE [35] belong to this category of work. Analogously, some tools such as Strider [15] or PeerPressure [29] adopt statistical techniques to compare values of configuration options in a problematic system with those in other systems to infer the root cause of a failure. These techniques require substantial effort to collect the baseline data.

VI. Conclusion and Future Work

The paper presents a practical technique 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.

Our experimental evaluation shows that the technique is highly effective in diagnosing misconfigurations. Moreover, it 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.

As we mentioned in the discussion, there exists limitations in our approach. In future we will develop a strategy to cope with cases that the value of a configuration option flows into dependent libraries. We also will exploit to diagnose failures caused by a combination of multiple configuration errors.

Acknowledgment

We thank Mohammadreza Ghanavati, Lutz Büch and Diego Costa in PVS group of Heidelberg University for their suggestions. We appreciate time and effort of the anonymous reviewers to appraise this work.

References

[1] 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.

[2] 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.

[3] 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, New York, NY, USA, 2010. ACM.

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

[5] Xiaoning Ding, Hai Huang, Yaoping Ruan, Anees Shaikh, and Xiaodong Zhang. Automatic software fault diagnosis by exploiting application signatures. In *Proceedings of the 22Nd Conference on Large Installation System Administration Conference*, LISA'08, pages 23–39, Berkeley, CA, USA, 2008. USENIX Association.

[6] Zhen Dong, Mohammadreza Ghanavati, and Artur Andrzejak. Automated diagnosis of software misconfigurations based on static analysis. In *IWPD 2013 at ISSRE*, pages 162–168, 2013.

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

[8] 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.

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

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

[23] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: Improving configuration management with operating system causality analysis. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 237–250, New York, NY, USA, 2007. ACM.

[24] 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.

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

[26] 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.

[27] T. Uchiumi, S. Kikuchi, and Y. Matsumoto. Misconfiguration detection for cloud datacenters using decision tree analysis. In *Network Operations and Management Symposium (APNOMS), 2012 14th Asia-Pacific*, pages 1–4, 2012.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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, New York, NY, USA, 2013. ACM.

[33] 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, New York, NY, USA, 2011. ACM.

[34] 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.

[35] 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.

[36] 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, New York, NY, USA, 2014. ACM.

[37] Sai Zhang and Michael D. Ernst. Automated diagnosis of software configuration errors. In *ICSE'13, Proceedings of the 34th International Conference on Software Engineering*, San Francisco, CA, USA, May 22–24, 2013.

[38] T. Zimmermann, R. Premraj, Nicolas Bettenburg, S. Just, A Schroter, and C. Weiss. What makes a good bug report? *Software Engineering, IEEE Transactions on*, 36(5):pages 618–643, Sept 2010.