ConfGuru - A System for Fully Automated Debugging of Configuration Errors

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

Matthias Iacsa
Institute of Computer Science
Heidelberg University, Germany
iacsa@stud.uni-heidelberg.de

Abstract—**Software applications routinely offer configuration settings to adapt them to specific deployment requirements. The number of available configuration options and their dependencies increase the likelihood of introducing configuration mistakes, with costly faults typically manifesting in a production environment. Automated diagnosis of configuration errors can help here, yet the practical value and acceptance of the proposed solutions depend - besides sufficient accuracy - on satisfying some non-functional requirements. These include: (i) low intrusiveness (i.e. little runtime data/instrumentation), (ii) full automation of the diagnosis process, and (iii) fast computation of a diagnosis.**

In this work we propose *ConfGuru*, an approach and a tool which attempts to fulfill all three of these requirements. *ConfGuru* complements and improves upon *ConfDoctor* [14], our previous (semi-automated) approach for diagnosis of configuration errors. *ConfGuru* adds a fast static analysis approach to identify all code locations where option values are read (so-called Option Read Points (ORPs)) in a targeted application. Previously, these locations needed to be found manually, severely limiting adoption of *ConfDoctor* for new targets. Furthermore, due to algorithmic optimizations we can reduce the total time for computing a diagnosis to below a minute, and streamline the analysis process.

Our evaluation shows that *ConfGuru* can diagnose configuration errors and extract ORPs from a variety of applications with an accuracy matching previous semi-automated approaches. Simultaneously, it offers fast adaptation to new target applications and well as full process automation, and has low response time. This makes *ConfGuru* suitable as a practical configuration error diagnosis tool or a service for real-world scenarios.

Index Terms—**Configuration errors, Automated debugging, Static analysis, Option read points**

I. Introduction

Most of today's non-trivial applications ship with a variety of configurable options, accessible to users via GUI or configuration files (plain text, XML, JSON, etc.). These serve the purpose of adapting the applications' behavior to user requirements without need to re-compile or re-build the code. Due to the multitude of complicated options that modern applications provide, configuration errors became one of the major causes of today's system failures [23], [29], [30]. Study [30] reveals that around one-third of high-severity user-visible failures are caused by configuration options.

Given this challenge, in recent years many tools and approaches for automated debugging of configuration errors have been developed [27], [21], [32], [33], [25], [20]. Unfortunately, most of the proposed solutions require comprehensive runtime data for diagnosis. This implies that code has to be instrumented, and potentially sensitive user data has to be disclosed. To address these shortcomings, our previous work [15], [14] proposes an approach and a tool called *ConfDoctor* capable of diagnosing misconfigurations in Java applications with only minimal runtime information and no need for error reproduction. Specifically, users only need to provide the stack trace of an error, and no other execution-related data. For a preprocessing phase, we require as input the program source code, and a list of locations where configuration options are read.

Unfortunately, ConfDoctor is a research prototype which can hardly be used for configuration diagnosis in real-world scenarios. The most severe shortcoming is the need to supply it with a complete list of the *Option Read Points* (*ORP*s), i.e. the code locations where option values are read from configuration files. Until recently, obtaining this data required manual code inspection for each new version of a targeted application, and thus had prohibitively high cost. As a consequence, ConfDoctor can diagnose configuration errors for only four specific application versions evaluated in [14], and it is hardly realistic that support for other applications/versions will be provided. Further shortcomings of ConfDoctor are a relatively long running time of the analysis for each new stack trace, and a complicated setup.

The first issue can be addressed via automated detection of ORPs, as proposed in [13], [11], [22]. While these approaches are highly accurate in extracting option names, they do not provide exact locations of ORPs. However, the latter aspect is of paramount importance for correctness of the diagnosis. Moreover, their long execution time make it necessary to pre-analyze all versions of a diagnosed application and to store the results in a database. This adds a significant overhead in terms of both time and space for a fully automated solution. Consequently, another solution tailored for efficiency and overall accuracy of diagnosis is needed.

To this end we propose *ConfGuru*, a tool for fully-automated configuration debugging, which is result of re-implementing ConfDoctor and complementing it with an efficient solution for detection of option read points. The latter component (called in this paper *ORPReader*) performs ORP detection based via static analysis implemented upon the WALA tool [28]. It is able to report the locations of ORPs which exactly correspond to the line numbers stated in the bytecode, as required by ConfDoctor. Moreover, compared to other tools like ORPLocator [13] (which uses static analysis on top of [8]), the execution time is be reduced from hours to seconds or minutes.

We also tackle the other shortcomings of ConfDoctor: its long running time, and a complicated setup. By improvements in implementation we are able to cut down the runtime of an analysis for a new issue to the order of one minute. Moreover, the diagnosis is streamlined by eliminating the need to pre-process the source code and storing the results in a database. Our evaluation shows that the accuracy of the diagnosis is similar to the case of manually supplied ORPs. We also show that ORPReader can retrieve a large fraction of the ORPs, and study the impact of various parameters on this coverage.

In summary, the contributions of this work are:

- We introduce *ConfGuru*, a tool for fully automated diagnosis of configuration errors in Java applications, which does not require adaptation to new applications or their versions.
- We propose *ORPReader*, a fast and accurate approach for extracting option read points of applications using static analysis, which is a component of the overall solution.
- We evaluate of the overall diagnosis accuracy of *ConfGuru*, and study the accuracy of ORPReader, in particular the impact of different call graphs on the number of retrieved ORPs.

In Section II we detail our approach and Section III discusses the implementation. In Section IV we evaluate our approach. Section V discusses related work, and Section VI states our conclusions.

II. Approach Description

This section describes the approaches and implementation choices used in ConfGuru and in ORPReader. As extraction of configuration options is technically a more advanced topic, we devote most of the space to this part.

A. ConfGuru: architecture and workflow

We integrate ORPReader and ConfDoctor into a stand-alone tool, ConfGuru. This required some significant re-implementation and additions to ConfDoctor, notably removing the dependency (and usage) of the DBMS MySQL. The workflow of the original ConfDoctor consists of two execution phases. In the first phase it is invoked with the main entry points of the application and externally specified ORPs as input. It completes the forward slicing, and saves the forward slices into the database. In the second phase the original ConfDoctor receives the error stack trace as input. It then computes the backward slices, writes these to the database, computes the chops via database query, and finally analyses the result.

We decided to remodel and streamline this workflow by calculating ORPs on the fly, keeping all intermediate results in memory. This removes the need for using a DBMS, significantly speeding up the analysis process. The new workflow is then the following one:

(i) Process command line arguments
(ii) Parse stack trace
(iii) Build call graph
(iv) Detect ORPs via ORPReader (see below)
(v) Perform forward slicing
(vi) Perform backward slicing
(vii) Perform chopping of both types of slices
(viii) Execute analysis and ranking
(ix) Present analysis results.

For steps (iii), (v), (vi) and (viii) we significantly changed ConfDoctor's implementation. However, we also introduced changes and improvements in other parts, e.g. extended the call graph construction by different entry point sets; introduced bug fixes and optimizations for the forward and backward slicing. We discuss some changes of Step (iii) below, while technical details of other changes compared to ConfDoctor can be found in [16].

1) Variants of call graphs: Step (iii) above computes a call graph of the targeted application which is needed for the forward/backward slicing techniques used in ConfDoctor [14]. To compute such a graph, an application entry point must be specified as its root. In ConfDoctor, a single entry point is specified manually. As an alternative, more application entry points can be considered (e.g. all methods `main()` found in the source code), and all the corresponding call graphs can be used in the analysis. This variant is also implemented and available in ConfGuru. It eliminates the need for manual specification of the entry point but incurs high computational cost. We evaluate the impact of each variant in Section IV.

The remainder of this section describes the approach of ORPReader for extraction of option read points.

B. Overview of ORPs extraction

Our overall workflow of extracting option read points (ORPs) consists of the following four steps:

1) Identify configuration class and its subclasses
2) Identify configuration API
3) Locate positions of ORPs
4) Infer option names at each ORP.

The first two steps require limited user input to specify a base configuration class and the pattern that the configuration API follows. Our method then identifies the full set of configuration classes and the concrete configuration API. The remaining two steps are fully automated.

Similarly to ORPLocator, we constrain ourselves to the configuration API of the configuration classes themselves, and do not infer additional methods like ConfAlyzer. Our approach diverges from ORPLocator in that we support more than just getter methods. Rabkin and Katz have found that the usual configuration API consists of parameterized getter methods [22] (e.g. Hadoop, HBase, FreePastry). Our examinations have found that configuration classes using fields as the API are also common (e.g. JChord, Randoop, Cassandra).

In the following we will discuss each of the steps in detail.

C. Selecting the configuration classes

Rabkin and Katz found that most applications use a single base class to manage their configuration options [22]. Our method assumes the existence of a single configuration class, which is the basis for managing all configuration options of the application. As the first step of the analysis, we find the configuration base class (as named by the user), and then identify all its subclasses.

We did not observe a consistent naming convention for these classes. For example, Apache Hadoop calls their class *Configuration* [3], Apache Cassandra and JChord prefer the name *Config* [2] [4], and Randoop's configuration class are named *GenInputsAbstract* [6]. Therefore we cannot reliably automatize finding this name, and assume that the user must provide the name of the configuration base class/interface.

When the base configuration class is known, all its subclasses can be identified from the application's class hierarchy. This is easy in Java, since it uses nominal subtyping. In WALA [28], the full set of configuration classes can be easily found by searching in the class hierarchy object (i.e. `callGraph.classHierarchy`) for subclasses or interface implementations of the base class/interface.

D. Identifying the configuration API

Field API. Some applications (JChord, Apache Cassandra, Randoop) use the fields of the configuration class to store and provide access to individual options. When this API is used, we assume that each field of each configuration class stores the current value of a particular option. In WALA, we can collect all such fields via the following simple code (in the Kotlin language):

```
for (confClass in confClasses) {
  confFields.addAll(confClass.allInstanceFields)
  confFields.addAll(confClass.allStaticFields) }
```

Method API. The other frequent case (Hadoop, HBase, FreePastry) is the configuration classes providing access to the options through parameterized getter methods, as already reported by Rabkin and Katz [22]. These methods follow a simple naming pattern (generally beginning with "get") and take a string as their first parameter. The value passed as this parameter indicates the name of the configuration option to be retrieved:

```
String someVariable=Config.getString("optionName");
```

For this API type we collect all methods which comply with these requirements: their name starts with *get*, they take a string as their first parameter, and they have a return value. We call them the *(option) getter methods*. The implementation on top of WALA is slightly more involved, because we have to handle instance and static methods differently due to their different JVM implementation.

Similarly to the name of the base configuration class, the user has to specify which type of API (field vs. method) is used by the diagnosed application. The following sections discuss each of the both API variants separately.

E. Locating the option read points

For field API. When extracting the ORPs we are only interested in the code points where the fields are used to *read* the option values. For this reason, we tag all statements as ORPs where a configuration field is used as a right-hand side value. With WALA, we can quickly find all such code points through the *memory access map* that corresponds to the *call graph* of the application. Given the enclosing call graph node and the instruction index of the read instruction, we can then extract the class name and line number of the ORPs location.

For method API. The method API is expected to be only used for reading the configuration options. Setting the options would usually be done via a corresponding set of "set*" methods. This means that every call to a *getter method* represents an option read point, and we tag all such statements as ORPs. Finding the call sites of methods is significantly more intricate than identifying field reads. The core mechanism to do so is the *call graph* provided by WALA. The nodes of the graph correspond to the methods of the application, and the (directed) edges indicate which methods call what others. To do the analysis on the call graph, we first need to find the graph nodes that represent our extracted *getter methods*.

```
var getterNodes = getterReferences.flatMap{
    callGraph.getNodes(it) }
```

In the call graph the nodes for a method that calls our *getter method* will have an edge pointing at the *getter method*'s node. Hence, we can find nodes containing call sites of the *getter methods* by asking the call graph for the predecessors of our getter nodes.

```
for (getter in getterNodes) {
  val predecessors = callGraph.getPredNodes(getter)
  // ...  }
```

However, this only gives us a broad knowledge of the call site's location. For now, we only know in which nodes and therefore methods the calls to the *getter methods* are located. For the ORPs to be useful to ConfDoctor and also to be able to continue with the inference of option names, we need to find the exact statement where the call happens. We continue by traversing through all call sites located in the predecessor node and checking which node it targets. Whenever we find a call site invoking our *getter method*, we can extract the concrete instructions where the call is made, and mark those as ORPs. Class name and line number are extracted in a similar way as in the case of the field API.

F. Inferring option names

The remaining task is to infer the names of the read options. For the method API. This is the most complex part of our approach, since the method's parameter (assumed to be the option name) might be an expression of multiple variables, or even dynamically computed. The results are reported together with the line number and the enclosing class of the option read point. Our tool requires no user input for this step.

For field API. If the configuration class exposes the options through its fields, then we take the field name as the option name. This is a simple approach, and typically produces the correct names. However, some option naming conventions are incompatible with Java syntax or naming conventions. Popular styles for option names include dots (e.g. *some.config.option*) or underscores (e.g. *some_config_option*). The former is illegal as a field name, while the latter violates the Java "camel case" convention.

Since it is straightforward to automatically transform between both forms, we ignore such differences in the evaluation (i.e. if an option is called *some_config_option* but we identify it as *someConfigOption*, we count that as a success).

For method API. When the configuration API consists of getter methods, we assume that the first (string) parameter of the method denotes the option name. Hence, we need to infer the possible values passed into the method at each call site. For this we propose the following recursive procedure to infer the values of an expression. The exact approach changes depending on the expression passed as a parameter, resulting in several cases.

The easiest cases are string literals (trivial) and static constants. In the latter case we obtain the exact value of the constant by recursively computing the values of the subexpressions until we descend to the base case (i.e. string literals). Other cases require more involved treatment, described below.

Other fields. If the expression references a field which is not declared `static final`, then inferring its value is not as simple. Essentially, in such a case we find all code points where the field is written to, and infer the value of each of the assigned expressions. We then report the possible values of the original expression as the union of the inferred values at each field write.

String concatenation. If the expression is a concatenation of other expressions, we recursively identify the possible values of those expressions and produce all concatenations of those values.

String.format call. Options names are also often dynamically built by using a call to `String.format`. Assuming that all parameters passed to the format method are strings, we infer its return value as follows: we infer the value of each expression passed to `String.format`, and then call `String.format` to obtain the resulting string value.

Local variables. When the expression refers to a local variable of the enclosing method, we recursively infer the value of the last write to that variable. If the last write cannot be determined, our method fails and returns an empty result.

Passed parameter. If the expression in question is itself passed into the enclosing method as a parameter, then we can not identify the possible values locally. Instead, we do a lookup of all locations where the enclosing method is called. For each of these call sites we infer the values of the expression passed as the corresponding parameter. Then we infer the values of the original expression as union of the values from each call site. One special case is when the enclosing method itself is a *getter method*. In that case we do not do any further inference, but instead discard the ORP altogether.

Other cases. There are a lot of cases not covered above. Virtually any expression that results in a string value can be passed to a *getter method*. Examples of these are: Arbitrary function calls (other than `String.format`), or values depending on conditionals (`if`, `for`, `?:`).

When we encounter such an expression, we stop the recursion and return an empty result set. This is a practical decision to limit the complexity of the method. Our examination of applications' code has shown that a vast majority of cases is still covered by previously described cases.

III. Implementation

In order to identify the ORPs we need to statically analyze the application's code. To this end we compared several mature frameworks offering capabilities to analyze the source code [7], or the bytecode [28], [1], [17]. We decided to use WALA [9], the *T.J. Watson Libraries for Analysis*. One of the reasons is that WALA is already used in the original implementation of ConfDoctor, which reduces dependencies on external libraries. Moreover, it provides all features needed for our implementation, namely class hierarchy analysis, pointer analysis, call graph construction, intermediate representation, and context-sensitive slicing.

For our implementation we used the relatively new JVM language Kotlin [5]. It is terser than Java, cutting down much of its usual boilerplate. Also, it offers a stronger type system, giving more guarantees regarding immutability, null-safety and generics. On top of that, Kotlin offers full Java interoperability, which allows for an easy integration with the legacy code and libraries in Java.

IV. Evaluation

In this section we evaluate our method by answering the following research questions:

RQ1 What is the diagnosis accuracy of ConfGuru compared to the original ConfDoctor results, with different variants of call graphs?

RQ2 What is the recall of the ORPReader in terms of reproducing the manually found option read points?

RQ3 What is the impact of call graphs and the classpaths on the results of ORPReader?

RQ4 Does the approach of ORP extraction generalize to other applications?

A. *Experimental setup*

Our approach is evaluated using real-world applications described in Table I. The user-specified input required by ConfGuru is listed in Table II. In particular, *API type* specifies whether application reads option values via method calls or via field accesses (data given by user).

Application	Version	Release Date	#Java Files	Java LOC
Cassandra	3.10	Nov 16, 2016	1,469	198,447
Hadoop	2.7.1	Jul 7, 2015	6,329	951,558
HBase	0.92.2	Jan 25, 2016	887	187,433
FreePastry	2.1	Mar 13, 2009	883	82,412
JChord	2.1	May 31, 2012	285	26,482
Randoop	1.3.2	Aug 23, 2010	233	20,360

Table I
APPLICATIONS USED IN THE EVALUATION. LOC COUNT ACCORDING TO THE TOOL *cloc*, HTTPS://GITHUB.COM/ALDANIAL/CLOC.

Application	Config. Base Class Name	API-Type
Cassandra	org.apache.cassandra.config.Config	Field
Hadoop	org.apache.hadoop.conf.Configuration	Method
HBase	org.apache.hadoop.hbase.HBaseConfiguration	Method
FreePastry	rice.environment.params.Parameters	Method
JChord	chord.project.Config	Field
Randoop	randoop.main.GenInputsAbstract	Field

Table II
USER-SPECIFIED INPUT REQUIRED BY CONFGURU (VERSION-INDEPENDENT).

We have to exclude (blacklist in WALA) several classes and packages from all our analyses, esp. the call three construction. This is necessary to prevent the scope of the analysis becoming too large, and due to crashes of WALA for some Java standard library functions. For details, see [16].

B. *RQ1: ConfGuru's diagnosis accuracy*

We first evaluate the accuracy of ConfGuru's configuration error diagnosis by comparing its results against those published in [14]. Table III shows the rankings of the actual root cause for 29 misconfigurations used in [14]. Notation x/y means that the true root cause (option) was listed at the position x of a ranked list with y suspicious options. The column "ConfDoc. Paper [14]" reports results from the original ConfDoctor evaluation. The column "Manual ORPs" shows the result of ConfGuru on manually extracted ORPs (supposedly the same as in [14]). There are some differences between these two columns (see e.g. Hadoop/3 and Hadoop/5). This can be attributed to changed implementation of ConfDoctor used in ConfGuru, but to the fact that the original ORP data used in [14] has been lost, and only a recreated (and possibly changed) dataset was used.

The columns "ORPReader * Graph" show results for the fully automated approach, i.e. with ORPs extracted automatically via ORPReader. The version "1-Entry Graph" uses a (small) call graph with entry point specified by the user, while "Full Graph" is the version where each method is considered as potential root of the call graph (this is a much larger call graph, see Section IV-D).

Overall, the accuracy of ConfGuru is comparable to the original approach using manually found ORPs (column "Manual ORPs") (for Randoop, the results are even identical). An exception is issue HBase/3, where the result of ConfGuru is considerably worse. This happens because a lot of “false positives” are added by ORPReader, while the weight of the actual root cause remains the same. In summary, automatic ORP extraction seem to produce diagnosis results of comparable quality as the baseline method.

Also, both the "1-Entry Graph" and "Full Graph" yield identical ranking of the root causes. Although the full call graph contains much more ORPs, the additional read points are most likely located outside the slicing graph. However, the only ORPs that actually influence ConfDoctor's results are those contained in the slicing graph. These relevant ORPs are also identified by ORPReader when the smaller "1-Entry Graph" is used. This is a nice finding, as it allows us a more efficient analysis. Our experiments show that using a smaller graph reduces the total running time by about a third.

C. *RQ2: Comparison to number of manually extracted ORPs*

In this section we compare the numbers of ORPs extracted by ORPReader against the numbers of manually found ORPs used in the evaluation of ConfDoctor [14]. This can give additional explanations of the fact described in Section IV-B that the accuracy of ConfGuru (i.e. with automated extraction of ORPs) matches the accuracy achieved in [14].

Table IV shows numbers of manually identified ORPs and options versus numbers of ORPs extracted by ORPReader using the most complete analysis scope (full call graph). The numbers of ORPs found for JChord and Randoop is lower for ORPReader. The reason for this is that both these applications use the fields of the configuration class to access the option values. In such cases, the manually gathered ORPs include not only the field reads, but also the field writes. For ORPReader we decided that only the field reads should be reported as ORPs.

ORPReader's higher number for Hadoop can be attributed to the fact that ORPReader is much more thorough than the manual analysis and finds a lot of complicated ORPs only reached by chained function calls and string concatenations.

The number of ORPs extracted for HBase is actually inflated. This is because HBase is not so much an autonomous application, and more a module for Hadoop. As such, it makes use of Hadoop's configuration class. Hence, ORPReader does not only identify all read points for HBase configuration options, but also for those of Hadoop.

D. *RQ3: Impact of the call graphs on ORPReader*

In this section we study the impact of the call graph alternatives on the coverage of ORPReader. To this aim we

Application	ConfDoc. Paper [14]	Manual ORPs	ORPReader 1-Entry Graph	ORPReader Full Graph
JChord	2 / 47	2 / 72	2 / 64	2 / 64
JChord	1 / 53	1 / 72	1 / 64	1 / 64
JChord	1 / 45	1 / 72	1 / 64	1 / 64
JChord	1 / 57	1 / 72	1-2 / 80	1-2 / 83
JChord	1 / 42	1 / 72	1 / 64	1 / 64
JChord	1 / 37	1 / 72	1 / 64	1 / 64
JChord	1 / 48	1-5 / 72	1-4 / 64	1 - 4 / 64
JChord	22 / 47	22-31 / 72	14-26 / 64	14 - 26 / 64
Randoop	1 / 37	1-2 / 41	1-2 / 47	1-2 / 47
Randoop	1 / 35	1 / 36	1 / 41	1 / 41
Randoop	1 / 47	1 / 50	1 / 55	1 / 55
Randoop	1 / 39	1 / 40	1 / 45	1 / 45
Randoop	1 / 41	1 / 40	1 / 45	1 / 45
Randoop	1 / 41	1 / 43	1 / 48	1 / 48
Randoop	4 / 43	4 / 47	4 / 52	4 / 52
Randoop	1 / 38	1 / 40	1 / 45	1 / 45
Hadoop	1 / 7	1 / 26	1 / 65	1 / 65
Hadoop	1 / 11	1 / 19	1 / 34	1 / 49
Hadoop/3	2 / 7	19-30 / 38	22-36 / 63	22-54 / 81
Hadoop	1 / 18	1 / 44	1 / 74	1 / 92
Hadoop/5	2 / 16	3-4 / 19	5-6 / 23	5-6 / 41
Hadoop	1 / 11	1 / 38	1 / 63	1 / 81
Hadoop	3 / 6	7 / 22	8 / 26	8 / 44
Hadoop	1 / 11	1 / 38	1 / 63	1 / 81
HBase	1 / 17	1 / 15	1 / 122	1 / 124
HBase	1 / 17	1 / 15	1 / 122	1 / 124
HBase/3	3 / 20	2 / 15	20-21 / 122	20-21 / 124
HBase	N	N / 12	N / 42	N / 42
HBase	3 / 5	2-3 / 12	2-3 / 42	2-3 / 42

Table III
CONFGURU'S DIAGNOSIS ACCURACY USING MANUALLY SPECIFIED VS. AUTOMATICALLY DETECTED ORPS, AND COMPARED WITH THE RESULTS PUBLISHED IN THE CONFDOCTOR PAPER [14]. THE NUMBERS SHOW THE RANKING OF THE TRUE ROOT CAUSE OUT OF ALL RANKED OPTIONS. FOR EXAMPLE, 2/64 MEANS THE ACTUAL ROOT CAUSE WAS RANKED FIRST OUT OF A TOTAL OF 64 CANDIDATES. AN N MEANS THAT THE TRUE ROOT CAUSE WAS NOT LISTED.

	Manual		ORPReader - Full	
Application	#ORPs	#Options	#ORPs	#Options
JChord	294	73	219	65
Randoop	211	58	176	61
Hadoop	195	141	580	404
HBase	186	91	935	679

Table IV
NUMBERS OF ORPS AND OPTIONS OBTAINED BY THE MANUAL ANALYSIS AND BY ORPREADER.

build call graphs of both Hadoop Common and Hadoop as a whole using different configurations. These only differ in regard to the classpath and entry points. Table V describes the different configurations used in the graphs' construction, and Table VI shows key data of the constructed graphs.

Table VII shows the number of retrieved options and option read points for each variation of the call graph. The reported times include both the extraction of option read points and the construction of the call graph. Building the call graph makes up more than 90% of the time. For comparison, we also show the same metrics reported for ORPLocator [13]. Note that the runtime of ORPLocator is significantly longer.

1) Impact of entry points: Assuming the same classpath, a comparison of the ORPs found when using different sets

Name	**Classpath**
Common	Top level contents of $HDS/common
Common + Libs	All contents of $HDS/common
Hadoop	Top level contents of $HDS/mapreduce, $HDS/hdfs, $HDS/common, $HDS/yarn
Hadoop + Libs	All contents of hadoop/share

Name	**Entry Points**
Single (Common)	org.apache.hadoop.fs.FsShell.main
Single (Hadoop)	org.apache.hadoop.hdfs.server. namenode.NameNode.main
Main	All `main()` methods of the application
Full	All methods of the application

Table V
VARIANTS OF THE CLASSPATHS AND THE ENTRY POINTS FOR BUILDING DIFFERENT VERSIONS OF THE CALL GRAPH (WITH $HDS = /HADOOP/SHARE/HADOOP)

of entry points shows that the results are strict subsets of one another. All ORPs found with the single entry point are found by using the main methods, and analogously all ORPs extracted using the *Main* graph are included in the output of the *Full* graph. This confirms our hypothesis that using more entry point for graph construction results in a more complete call graph, and strictly better results.

However, this comes at a significant runtime cost. Analysis time when using the *Full* entry point set is usually about 10 times higher than that of the *Main* set and in the extreme 150 times higher that the *Single* set (analyzing all of Hadoop including the libraries). If the goal is to maximize the extracted ORPs, then using a full set of entry points is the way to go. However, as shown in Section IV-B, single entry points are sufficient to achieve high diagnosis accuracy.

2) Impact of the classpath: Including libraries in the classpath increases the number of retrieved options and ORPs. We have investigated the reasons for this somehow surprising finding and found two causes.

Construction	Single		Main		Full	
	#Nodes	#Edges	#Nodes	#Edges	#Nodes	#Edges
Common	1,676	8,781	3,726	34,988	22,338	706,875
Common + Libs	4,966	58,925	23,416	708,343	61,250	4,472,073
Hadoop	7,787	113,042	28,578	916,183	95,052	6,656,268
Hadoop + Libs	30,263	659,611	126,669	13,267,099	264,335	36,254,607

Table VI
GRAPH SIZES FOR DIFFERENT ENTRY POINT SETS.

Entry Point	Libs in CP?	Common #ORPs	Common #Options	Common Time	Hadoop #ORPs	Hadoop #Options	Hadoop Time
Single	No	15	15	3s	259	217	8s
Single	Yes	16	16	6s	260	218	27s
Main	No	56	48	4s	1,284	878	26s
Main	Yes	62	54	19s	1,416	958	5m15s
Full	No	412	273	53s	2,564	1,477	10m10s
Full	Yes	613	377	3m20s	3,202	1,854	1h10m
ORPLocator		261	210	70m	1,861	1,300	7h48m

Table VII
NUMBER OF OPTION READ POINTS AND UNIQUE OPTIONS EXTRACTED BY ORPREADER USING DIFFERENTLY CONSTRUCTED CALL GRAPHS ("CP" MEANS CLASSPATH). SAME METRICS ARE REPORTED FOR ORPLOCATOR [13] FOR COMPARISON.

Application	#Extracted	#Documented	#Documented ∩ Found	Percentage
Apache Cassandra	172	147	147	100%
FreePastry	203	206	187	90.8%

Table VIII
NUMBER OF EXTRACTED AND DOCUMENTED OPTIONS FOR ADDITIONAL APPLICATIONS.

Libraries contain additional ORPs. Some of the additional files included in the *Common + Libs* and *Hadoop + Libs* scopes actually contain ORPs. A good example is `hadoop/share/tools/lib/hadoop-aws-2.7.1.jar`, which loads a lot of the S3 options of Hadoop Common.

The dependency problem. This effect is a lot more subtle. While WALA tries to include all classes in the Jar files given by the classpath, sometimes classes are missing from the resulting class hierarchy and call graph. To analyze a class, WALA needs the complete information about it, including all superclasses and implemented interfaces. When one of these is missing, the subclass is excluded from the analysis, and all its methods are missing in the call graph. This shows that exhaustive results can only be obtained if all dependencies are included in the classpath of the analysis.

For example, Hadoop features the classes `oncrpc.RpcProgram` and its subclass `hdfs.nfs.nfs3.RpcProgramNfs3`. The latter contains ORPs and is thus relevant for our analysis. However, both these classes are missing in the *Hadoop* call graph, despite of being included in the classpath. The reason for this is that `RpcProgram` extends `SimpleChannelUpstreamHandler`, which is a class defined in a library (`netty-3.6.2.Final.jar`) not included in the analysis classpath. Since a superclass of `RpcProgram` is missing, WALA cannot analyze it and drops it from the analysis. Transitively the same happens with `RpcProgramNfs3`.

E. RQ4: Extraction of ORPs from other applications

All results stated so far were obtained for the same applications as in the ConfDoctor work [14]. Here we want to study how well ORPReader generalizes to other applications, and thus fulfills the promise of effortless adaptation to other software. Table VIII shows the results.

Apache Cassandra 3.10. ORPReader successfully extracts all documented options for Apache Cassandra (these options were compiled from the documentation web page [2]).

FreePastry 2.1. The FreePastry documentation of configuration options was extracted from the *freepastry.params* file included in the application. A total of 19 options were documented but not extracted by ORPReader. We were unable to find any references to 18 of these options during a manual inspection of the source code. One option (`pastry_factory_bootsInParallel`) had an ORP, which was however commented out. We conclude that our tool did not miss any relevant options.

V. Related Work

Related work falls into two categories: approaches for configuration error diagnosis, and techniques for extracting configuration options.

A. Configuration Error Diagnosis

In recent years, configuration-related issues have attracted attention of researches, culminating in empirical studies [29], [23], [24], and a variety of diagnosis approaches [21], [32], [27], [33], [12], [25], [20]. Latter can be classified into two areas: program analysis and non-program analysis. Since the former domain is closer of our work, we refer reader to [14] for an overview of works in the non-program analysis.

Program analysis. ConfAid [10] applies dynamic information flow analysis techniques. Sherlog [31] uses static analysis to infer the execution path and state at runtime. ConfAlyzer [22] uses static data flow analysis to detect suspicious options. ConfDiagnoser [33] uses dynamic analysis technique to compare runtime profiles for each option. SPEX [29] analyze source code to identify configuration constraints and detects their violations. [25] diagnoses configuration errors in complex applications composed of multiple components (e.g. LAMP-stack, i.e. Linux, Apache Web Server, MySQL, PHP). It uses a combination of static analysis (mostly backward slicing) and information on manually determined configuration dependencies between components. In [20] authors propose to use symbolic taint analysis to identify how options and their values are determine which code regions are executed. This information can be used for error diagnosis.

B. Extraction of Configuration Options

The closest work to ORPReader is ConfAlyzer [22]. This work uses static analysis to extract a list of configuration options from source code. Since ConfAlyzer uses source code as input, the identified ORP locations do not match those required by ConfDoctor which works with the bytecode. Contrary to this, the approach presented in this paper provides exact ORP data for ConfDoctor, and thus allows integration of both tools.

ORPLocator [13] uses srcML [8] to parse the source code into XML format. Similarly to ORPReader, it first identifies all subclasses of a user-specified configuration base class. In the following steps methods and call sites reading option values are identified. After inferring option names it builds a map between these names and the corresponding ORPs. Unfortunately, ORPLocator has large runtime (see Table VII), significant memory requirements, and contrary to ORPReader it does not allow to trade accuracy for execution time. Finally, for the reasons outlined in Section I (e.g. mismatch of line numbers for ORP locations) it cannot be used together with ConfDoctor.

PrefFinder [18] proposed by Jin *et al.*, uses static and dynamic analysis techniques to extract configuration options. The SCIC [11] extends ConfAlyzer to handle options in the key-value model and the tree-structured model. Works [26][19] analyze configuration variant and space in configurable system software.

VI. Conclusion and Future Work

We have have proposed *ConfGuru*, an automated debugger for diagnosing configuration errors in Java applications. It is based on improved version of ConfDoctor [14] and a new, computationally efficient approach and a tool for extracting option read points via static analysis. ConfDoctor combines low intrusiveness (error stack trace as the only runtime data) with fast diagnosis speed and the ability to handle any Java application without manual adaptation. This makes our approach highly relevant in practice. We are currently working on offering ConfGuru as a web service to a wider audience.

In our study we discovered that the primary reason for missing ORPs are deficiencies of the external WALA tool [9]. This problem has also affected ConfDoctor before, but has first been uncovered by our evaluation. Our future work will target finding workarounds or solutions for the underlying *dependency problem*, as it is of paramount importance for improving the accuracy of both ORPReader and ConfDoctor.

References

[1] ASM. http://asm.ow2.org/. Accessed: 2018-07-20.

[2] Cassandra Documentation: configuration. http://cassandra.apache.org/doc/latest/configuration/cassandra_config_file.html. Accessed: 2018-07-20.

[3] Hadoop Documentation: configuration class. https://hadoop.apache.org/docs/r2.7.3/api/org/apache/hadoop/conf/Configuration.html. Accessed: 2018-07-20.

[4] JChord Documentation: configuration class. http://pag-www.gtisc.gatech.edu/chord/javadoc/chord/project/Config.html. Accessed: 2018-07-20.

[5] Kotlin Programming Language. https://kotlinlang.org/. Accessed: 2018-07-20.

[6] Randoop Documentation: configuration class. https://randoop.github.io/randoop/api/randoop/main/GenInputsAbstract.html. Accessed: 2018-07-20.

[7] Spoon. http://spoon.gforge.inria.fr/. Accessed: 2018-07-20.

[8] srcML. http://www.srcml.org/. Accessed: 2018-07-20.

[9] T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/Main_Page. Accessed: 2018-07-20.

[10] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI'10*, pages 1–11, Berkeley, CA, USA, 2010. USENIX Association.

[11] F. Behrang, M. B. Cohen, and A. Orso. Users beware: Preference inconsistencies ahead. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 295–306. ACM, 2015.

[12] F. Behrang, M. B. Cohen, and A. Orso. Users Beware: Preference Inconsistencies Ahead. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 295–306, New York, NY, USA, 2015. ACM.

[13] Z. Dong, A. Andrzejak, D. Lo, and D. Costa. ORPLocator: Identifying read points of configuration options via static analysis. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 185–195, Oct 2016.

[14] Z. Dong, A. Andrzejak, and K. Shao. Practical and accurate pinpointing of configuration errors using static analysis. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 171–180. IEEE, 2015.

[15] Z. Dong, M. Ghanavati, and A. Andrzejak. Automated diagnosis of software misconfigurations based on static analysis. In *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on*, pages 162–168. IEEE, 2013.

[16] M. Iacsa. Automated configuration debugging via static analysis. Master's thesis, Ruprecht-Karls-Universität Heidelberg, 2017.

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

[18] D. Jin, M. B. Cohen, X. Qu, and B. Robinson. Preffinder: Getting the right preference in configurable software systems. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 151–162, New York, NY, USA, 2014. ACM.

[19] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM.

[20] M. Lillack, C. Kästner, and E. Bodden. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering*, pages 1–1, 2017.

[21] A. Rabkin and R. 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.

[22] A. Rabkin and R. Katz. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 131–140. ACM, 2011.

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

[24] M. Sayagh, Z. Dong, A. Andrzejak, and B. Adams. Does the Choice of Configuration Framework Matter for Developers? Empirical Study on 11 Java Configuration Frameworks. In *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017*, pages 41–50. IEEE Computer Society, 2017.

[25] M. Sayagh, N. Kerzazi, and B. Adams. On Cross-stack Configuration Errors. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 255–265, Piscataway, NJ, USA, 2017. IEEE Press.

[26] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 47–60, New York, NY, USA, 2011. ACM.

[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] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 244–259, 2013.

[30] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, 2011.

[31] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. *SIGARCH Comput. Archit. News*, 38(1):143–154, Mar. 2010.

[32] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. 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.

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