2016 IEEE 9th International Conference on Cloud Computing

# CSMiner: An Automated Tool for Analyzing Changes in Configuration Settings across Multiple Versions of Large Scale Cloud Software

Nhan Nguyen, Mohammad Maifi Hasan Khan and Kewen Wang
*Department of Computer Science and Engineering*
*University of Connecticut*
*Email: nhan.q.nguyen@uconn.edu, maifi.khan@engr.uconn.edu, kewen.wang@uconn.edu*

***Abstract*—As software evolves, the number of configuration settings and their usage scenario often change as well, causing system misconfiguration and performance degradation. However, no tool exists today that can aid system administrators/developers answering questions such as *"What are the new configuration settings in this new version?"*, or *"Where and How is setting X used in the new version?"*. As manually investigating answers to these questions is almost impossible due to the number of settings and size of the software, this paper investigates the design of an automated tool (CSMiner) leveraging static program analysis techniques that helps users to understand how and where a particular setting is used in a program and how settings have evolved across different versions of a software system. CSMiner was applied on four different open source software packages, namely, Apache Cassandra, ElasticSearch, Apache Hadoop, and Apache HBase, and CSMiner identified 109 (out of 109), 109 (out of 113), 811 (out of 847), and 160 (out of 167) settings for these software packages respectively. In each case, CSMiner successfully identified the changes in configuration settings across multiple versions with high accuracy.**

***Keywords*-Configuration Analysis; Characteristic Study; Configuration Evolution; Cloud Platform**

## I. Introduction

Modern software often offers a large number of configuration settings that may be tuned to adjust software behavior or to improve performance [1]. However, as software keeps evolving, the number of configuration settings and their usage scenario often change as well (e.g., addition/removal of configuration settings) [2], making tuning settings extremely challenging. As reported in [1], 16.7%∼32.4% of the misconfigurations happen due to system upgrade, reconfiguration, and/or other tasks that cause changes in complex systems. Unfortunately, when system administrators/developers are faced with questions such as *"What are the new configuration settings in this new version?"*, or *"Where and How is setting X used in the new version?"*, they primarily have to rely on online resources and software documentation to find answers to such questions [3], [4]. However, such resources often do not document changes in configuration settings across multiple versions, making it hard to understand the full implication of upgrading a software package to a different version.

As investigating answers to the aforementioned questions manually can be tedious and often error prone, this paper investigates the design of an automated tool leveraging static program analysis techniques that can help users understand how settings have evolved across different versions of a software package and how it may affect the execution flow of the software. Specifically, the presented tool can assist in answering the following questions: (a) Where and how setting X is used in my program?, (b) Which configuration settings have been added or removed in the new version?, and (c) Which configuration settings have been modified in the new version and how?. To answer question (c), we consider two types of changes, namely, changes in the set of methods and/or fields that use a setting, and changes in the way a setting influences the execution of a program.

To answer the aforementioned questions, in this paper, we present CSMiner, an automated tool that can analyze a program, and identify changes in load-time configuration settings with high accuracy, along with identifying the type of settings and usage location(s) in the program. To implement the tool, first, we extend the open source tool Soot [5] to extract the call graphs from a program that identifies the execution path(s) that lead to the location in the program where a particular setting is loaded, which we call configuration-oriented call graph in this paper. Subsequently, we used the idea of taint analysis [6] to identify parts of the code that may be affected by a particular setting, which is represented as a configuration-oriented data-flow graph. In this paper, we use the term configuration-oriented graph (CoG) to refer to the configuration-oriented call graph and configuration-oriented data-flow graph. Finally, once the tool constructs the CoG for a particular setting for different versions of a software package, the tool uses graph comparison techniques to compare the resulting graphs to detect changes that may have been made to that setting across multiple versions.

To evaluate the performance of the tool, we applied our tool on different versions of four open source large-scale software packages, namely, Apache Cassandra [7], ElasticSearch [8], Apache Hadoop [9], and Apache HBase [10]. In each case, the tool successfully identified the information of configuration settings. Specifically, CSMiner identified 109 (out of

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.67

109), 109 (out of 113), 811 (out of 847), and 160 (out of 167) settings for Apache Cassandra (version 2.1.8), ElasticSearch (version 2.1.1), Apache Hadoop (version 2.7.1), and Apache HBase (version 1.1.1) respectively. Furthermore, CSMiner successfully identified the number of added settings for all of these software with a false negative rate between 0% to 7.5% and the number of removed settings with a false negative rate between 0% to 9.5%.

The rest of the paper is organized as follows. Prior efforts that are related to our work are discussed in Section II. Design and implementation of the tool are presented in Section III. Section IV presents the result of our study. The limitations of our current work are discussed in Section V. Finally, Section VI concludes the paper.

## II. Related Work

As the number of configuration settings continues to grow, analyzing and understanding the influence of these settings on a program is becoming increasingly important. Not surprisingly, researchers have investigated various approaches that may help users understand various aspects of configuration management and tuning in complex large-scale systems, including misconfiguration detection [11], [12], troubleshooting [13], [14], [15], [16], [17], and performance tuning [18], [19]. The problem of misconfiguration troubleshooting in commercial and open source systems has been studied as well [1].

Among numerous efforts, interestingly, only a handful of recent work focuses on understanding the influence of configuration settings on programs. For example, one recent work applied static source code analysis technique to identify code fragments that depend on certain configuration settings in Android applications [20]. Another work explored the idea of static extraction of program configuration settings [21] to categorize configuration settings into different groups based on their types. One of the recent works, that is closest in spirit to our work, manually studied real-world configuration changes in large scale software systems [2].

In contrast to prior efforts, our work aims to help system administrators/developers to gain insights regarding the evolution of configuration settings in software systems leveraging static source code analysis techniques [22], [23], [5] such as call graph construction and taint analysis by automatically identifying changes in configuration settings across multiple versions, and complements prior efforts that focus on identifying the type of configuration settings [21] and attempt to explain which code fragments depend on which configuration settings [20].

## III. Approach

In this section, we first introduce the necessary background concepts regarding configuration-oriented graph, and then describe the design and implementation details of our tool. The details are below.

### A. Background

In our work, to study the evolution of configuration settings across multiple versions, we define a configuration-oriented graph (CoG) as a directed graph that contains two parts, namely, the Configuration-oriented Call Graph (CCG) and the Configuration-oriented Data-Flow (CDF) graph.

The Configuration-oriented Call Graph (CCG) for a setting "x" includes all execution paths from starting points of a program to all the methods in which configuration setting "x" gets loaded (i.e., configuration setting loading method). In other words, CCG for a setting "x" is a subgraph of the call graph for the whole program and only contains methods related to configuration setting "x". CCG can be used to answer the question *"Where is setting X loaded in my software and what paths lead to that location from starting points?"*. In this work, starting points include the entry points of a program (e.g., main function), and any other methods which can be invoked externally (e.g., through API) and can reach the loading method of setting "x". Additionally, if a configuration setting is assigned to a field of a class, we consider the class's constructor as a loading method as well. To increase the accuracy of the graph, we also consider all methods that use the assigned field of a setting as the loading methods and generate paths from starting points to those methods.

The Configuration-oriented Data-Flow graph (CDF), on the other hand, represents the execution flow after a setting "x" gets loaded in the program. CDF assists users to identify the code fragments that depend on a configuration setting, and helps answer questions such as *"Where and How is setting X used in my software?"* (e.g., which classes and which methods use setting X) or *"What is the role of setting X in my software?"* (e.g., setting X is used as a control variable, or as a parameter of a method).

```
01: public static void main(String[] args) {
02:     runTaskA();
03: }
04: public static void runTaskA() {
05:     Configuration conf = new Configuration();
06:     int nbStrings = conf.getInt("settingA", 1);
07:     String[] array = new String[nbStrings];
08:     boolean isCorrect = conf.getBoolean("settingB", true);
09:     if (isCorrect) {
10:         performJob();
11:     }
12: }
```

Figure 1: A simple example demonstrating the use of configuration settings. The configuration read functions *getInt* and *getBoolean* are borrowed from Apache Hadoop.

A simplified example of a CoG (that includes both CCG and CDF) is shown in Figure 2, and the corresponding source code is illustrated in Figure 1. The CCG includes the execu-

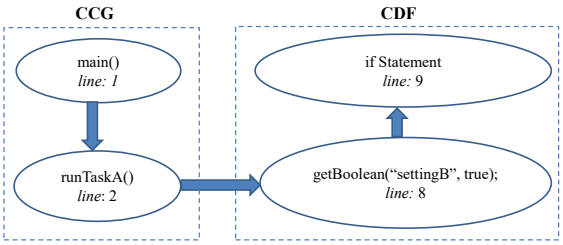


Figure 2: The CCG of the CoG (left) and the CDF of the CoG (right) for the configuration setting "settingB".

tion path from function *main()* (line 01), which is the starting point of the program, to function *runTaskA()* (line 04), which is the loading point of two configuration settings, namely, "settingA" (line 06) and "settingB" (line 08). The CCG in this case will be same for these configuration settings. As these settings affect the execution differently once loaded during program execution, the CDFs for these two settings are going to be different. Due to space limitation, we only show the CDF for "settingB" in Figure 2 where this setting is used as a conditional variable in an if statement.

### B. Constructing the Configuration-oriented Graph

Before the tool can construct the Configuration-oriented Call Graph (CCG) for a given software package, we first need to generate the list of configuration read functions (i.e., configuration loading APIs) and provide that list as input to the tool. To understand how configuration settings are loaded in software, we manually studied 10 open source software packages (listed in Table I) and identified several approaches that are commonly used as follows.

| Approach | Software Name |
|---|---|
| Approach 1 | Cassandra, Voldemort[1] |
| Approach 2 | Hadoop, HBase[2], Chukwa[3], Derby[4], Flume[5] |
| Approach 3 | ElasticSearch[6], JMeter, CloudStack[7] |

Table I: Table of open source software and their configuration loading approaches.

*Approach 1*: One of the common approaches for loading configuration settings in a program is where each setting has its own **get** and **set** function or a pair of **get** and **set** function for a category of settings. An example of this approach is shown in Figure 3. In this example, the configuration setting "preheat_kernel_page_cache" is read to the variable *preheat_kernel_page_cache* of class *Config* and is accessible through the API *shouldPreheatPageCache()* of class *DatabaseDescriptor*.

[1]http://www.project-voldemort.com/voldemort/
[2]https://hbase.apache.org/
[3]http://chukwa.apache.org/
[4]http://db.apache.org/derby/
[5]http://flume.apache.org/
[6]https://www.elastic.co/products/elasticsearch
[7]https://cloudstack.apache.org/

*Configuration name:* "preheat_kernel_page_cache"

*Class org.apache.cassandra.config.Config*
public boolean preheat_kernel_page_cache = false;

*Class org.apache.cassandra.config.DatabaseDescriptor*
public static boolean **shouldPreheatPageCache**()
{
return conf.**preheat_kernel_page_cache**;
}

*Class org.apache.cassandra.io.sstable.SSTableReader*
if (DatabaseDescriptor.**shouldPreheatPageCache**() && fd > 0)
CLibrary.preheatPage(fd, entry.getValue().position);

Figure 3: Example configuration setting in Cassandra.

*Approach 2*: In this approach, an API is used to retrieve all settings that share a common property such as type (e.g., integer). For example, in Figure 4, the setting "hadoop.security.group.mapping.ldap.ssl" has type boolean and is loaded by an API named *getBoolean()* which has the following parameters: name of the setting and the default value.

*Configuration name:* "hadoop.security.group.mapping.ldap.ssl"

*Class org.apache.hadoop.security.LdapGroupsMapping*
public static final String LDAP_CONFIG_PREFIX =
"hadoop.security.group.mapping.ldap";
…
public static final String LDAP_USE_SSL_KEY = LDAP_CONFIG_PREFIX + ".ssl";
…
useSsl = conf.**getBoolean**(LDAP_USE_SSL_KEY, LDAP_USE_SSL_DEFAULT);

Figure 4: Example configuration setting in Hadoop.

*Approach 3*: Finally, the third approach is to use the Java built-in Properties class to load settings. For example, in case of JMeter [24], two methods named *getPropDefault()* and *getProperty()* are used to read configuration settings. In particular, the method *getPropDefault()* is a polymorphic method which has format *Type getPropDefault(String propertyName, Type propertyType)* in which Type can be int, boolean, long, or String.

Once we generate the list of configuration loading APIs and provide it as input to the tool, to construct the CCG for each individual setting, the tool first identifies the configuration loading points for a particular setting. Next, for each of these configuration loading points, the tool finds all the methods that invoke it, and then recursively back tracks until it reaches the starting point(s) of the program. Please note that, there can be multiple such CCGs for each setting as the same setting may be loaded in different parts of a program.

Once the CCG is constructed, next, to construct the configuration-oriented data-flow graph (CDF) which is a directed graph, the tool considers each configuration setting loading point as the source (i.e, indegree equals 0) and constructs one CDF starting from each of the loading points. For instance, if there are two different points in the software where setting "x" gets loaded, there will be two different CDFs for "x". As a configuration setting can be assigned

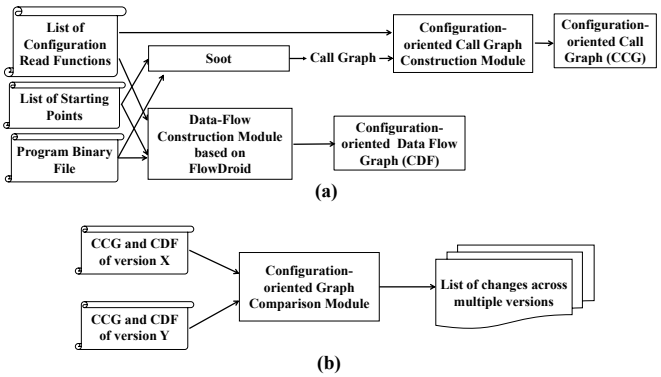


Figure 5: The architecture of CSMiner.

to a variable, a field, or passed to a method as a parameter, the tool considers all these possibilities while constructing the CDF. The sinks (i.e., outdegree equals 0) of the graph are determined based on how we describe the flow functions for each individual flow edge. In the scope of this work, to keep the size of the CDF tractable, we do not explore all transitive effects of configuration settings, and only retrieve information regarding how a configuration setting is used within a program. Specifically, when a configuration setting is passed as a parameter of a method that belongs to a third party library, we do not explore how it is used within that library. However, if a setting is passed as a parameter of a method that belongs to the program being analyzed, we do explore further.

### C. Implementation of the Tool

The architecture and the execution flow of the tool is illustrated in Figure 5. As shown in the figure, the tool first builds the CoG (Figure 5 (a)) for each setting for each version of a software package and then compares the constructed CoGs (Figure 5 (b)) to identify the changes regarding configuration settings of that software package across different versions. In Figure 5 (a), input to the tool includes the list of configuration loading APIs, the list of starting points, and the program binary files. These configuration read functions are usually "well-defined" functions as explained in the previous subsection (III-B) and also discussed in [21]. The starting points are usually entry points or public APIs which are available online (e.g., Hadoop[8]). Next, once the graphs are generated by the modules in Figure 5 (a), the Configuration-oriented Graph Comparison Module (Figure 5 (b)) reads and compares them to output the list of changes. The implementation details are described as follows.

***Configuration-oriented Call Graph Construction Module.*** To implement the module that constructs the CCG, we leverage Soot [5], which is an open source framework for analyzing Java source code or Java bytecode. Due to this flexibility, CSMiner can be easily applied to programs written in other languages such as Scala (e.g., Apache Spark[9]) that are compiled to Java bytecode. In Soot, we can use a method's signature to query information related to that method from the call graph. Specifically, we can use a target method's signature to get the list of methods that call that method. Using this feature of Soot, CSMiner first constructs the call graph as a bi-directional graph, and then applies the Depth-First Search algorithm starting from the setting loading points to traverse backwards to identify the starting points of the program. In our study, each of the software has multiple starting points (e.g., more than a dozen *main()* functions for Hadoop and HBase). Also, the software we analyzed contain significant number of public methods such as remote procedure calls (RPCs), which may not be reachable from a given set of starting points but can be accessed by external invocations. To deal with this case, we applied the idea described in [13]. In short, CSMiner generates Java code that instantiates an object of the corresponding classes of those methods in the *main()* function. It then invokes these public methods by the created object. There are cases where a method *m_1()* of the host program is called within another method *m_2()* and the method *m_2()* is used by external libraries. For example, a method is called by method *run()* of class *Thread* and the method *run()* is used by other functions in the same program or in external libraries. To avoid the explosion of the graph, we eliminate all calls made by external libraries. As all methods in a program have the common prefix in their signature, we use that prefix to detect whether a method belongs to the host program or not. For example, we use the prefix "org.apache.cassandra" for Apache Cassandra and "org.apache.hadoop" for Hadoop. The constructed call graph is the CCG which the tool can output by applying the Depth-First Search algorithm which begins from starting points to traverse to all loading points.

***Data-Flow Construction Module.*** To implement the module that constructs the CDF, we leverage the data-flow tracking component FlowDroid [6] framework for performing taint analysis for Java bytecode programs, which implements the IDFS/IDE framework [22] for data-flow analysis for Java using Heros [23]. Specifically, for a given list of source and sink nodes, FlowDroid outputs a list of possible paths from source to sink nodes. In particular, in case of taint analysis, if we know the source and the sink node, then FlowDroid can help us find whether a path exists from the source to the sink node or not. However, when we build a CDF, as we only know the "source" node, which is the loading point of a configuration setting, we extended FlowDroid by applying the data-flow analysis starting from the source nodes and continue the analysis along

[8]https://hadoop.apache.org/docs/r2.7.1/api/

[9]http://spark.apache.org/

all possible edges as long as there is a target node which can be reached from the source node. We implemented a dedicated class which receives all taints propagated by the IDFS problem solver, and analyzes these taints to identify different kinds of statements (e.g., if statement, invocation statement) as well as which statements are related to the setting being analyzed. CSMiner then applies the Depth-First Search algorithm on the constructed inter-procedural control flow graph to output the CDF.

### *D. Identifying Changes in Configuration Settings across Multiple Versions*

To detect configuration settings that are added, removed, or modified in the new version of a software package, first, we use CSMiner to extract the list of configuration settings for each version by analyzing the program binary file. For example, in case of Hadoop, CSMiner determines the value of the first parameter of the **get** function to get the name of the passed configuration setting. As shown in Figure 4, the name of the configuration setting "hadoop.security.group.mapping.ldap.ssl" is actually the value of the String LDAP_USE_SSL_KEY. Please note that, if the names of settings are dynamically constructed as shown in Figure 4, the constant propagation mechanism in Soot helps to get its full name.

```
   input: CoG1, CoG2 of a configuration setting
   output: List of modifications made to the configuration
           setting
1  Function compareCoG
2      Report all starting points appear in CCGs of CoG1
         but do not appear in CCGs of CoG2 and
         otherwise;
3      foreach source s in CDF1 of CoG1 do
4          if ∃ a CDF2 of CoG2 has same source s then
5              compareCDF(CDF1, CDF2, s);
6          else
7              report the difference at source s;
8          end
9      end
10 end
```

**Algorithm 1:** Configuration-oriented graph comparison algorithm.

Once we extract the name of the settings, we compare the lists of names from the old and new version, and have three cases to consider as follows. *First,* if a name appears in the list of the new version and does not appear in the list of the old version, then it is considered as a newly added configuration setting. This will help to answer the question *"What are the new configuration settings in this new version?"*. *Second,* if a name appears in the list of the old version and does not appear in the list of the new version, then it is considered as a removed configuration setting. *Finally,* the names that are common across the old and new version (*the common list*), those represent both the unmodified and the modified configuration settings. To answer questions such as *"Which configuration settings are modified?"* and *"How are configuration settings modified?"*, we compare the configuration-oriented graphs for each setting in *the common list* for both versions of a program. The algorithm for graph comparison is illustrated in Algorithm 1. Two parts of a CoG (i.e., CCG and CDF) are compared separately as follows. *First,* we compare the CCGs from two different versions to find whether execution paths leading to a configuration setting are being added/removed in the new version (line 2). *Next,* we compare the CDFs to identify whether there is any change regarding how a setting may affect the execution flow of the software. The algorithm that compares the CDFs is shown in lines 3-9 of Algorithm 1 and in Algorithm 2. The algorithm is based on the fact that, if a path (i.e., data-flow) appears in the graph of the old version but does not appear in the graph of the new version, then the corresponding use of the setting has been removed in the new version. In contrast, if a path appears in the graph of the new version but does not appear in the graph of the old version, then a new usage scenario has been added in the new version. Please note that, in our work, a change in the name of methods is also viewed as a modification in the usage scenario.

```
   input: CDF1, CDF2 of the configuration setting, s
   output: List of modifications made to the configuration
           setting on the data-flows
11 Function compareCDF
12     P1 ← all paths from the source s to sinks of CDF1;
13     P2 ← all paths from the source s to sinks of CDF2;
14     foreach path p1 in P1 do
15         if ∃ a path p2 in P2 has same sink with p1 then
16             check the common internal nodes and
                 report the difference (if any);
17         else
18             report the difference;
19         end
20     end
21 end
```

**Algorithm 2:** Configuration-oriented data-flow graph comparison algorithm.

## IV. Evaluation of the Tool and Empirical Result

We applied CSMiner to study the evolution of configuration settings in four large scale open source software packages. All experiments were conducted on a PC with i5-3450 CPU and 16 GB of RAM. As multiple versions are available for the software listed in Table II, and all of them are shipped with large number of configuration settings, we consider them suitable for our study. While we studied

| Program | Years | LoC | Versions | Boolean | Integer | Long | Double | Float | String | Other | No. of settings |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Cassandra | 2 | 106,885 | 1.2, 2.0, 2.1 | 12 | 45 | 11 | 3 | 0 | 25 | 13 | 109 |
| Elastic | 1 | 273,416 | 1.5, 1.7, 2.1 | 14 | 19 | 4 | 2 | 0 | 42 | 28 | 109 |
| Hadoop | 2 | 1,386,829 | 2.2, 2.6, 2.7 | 94 | 295 | 156 | 3 | 26 | 181 | 56 | 811 |
| HBase | 1 | 732,522 | 0.90, 0.98, 1.1 | 32 | 59 | 21 | 3 | 19 | 17 | 9 | 160 |

Table II: The software systems that were studied and their characteristics. Column "LOC" lists number of lines of code.

| Program | No. of documented settings | False Negative (%) |
|---|---|---|
| Cassandra | 109 | 0 |
| Elastic | 113 | 3.5 |
| Hadoop | 847 | 4.3 |
| HBase | 167 | 4.1 |

Table III: Percentage of settings missed by the tool.

multiple versions of each of these software in our study, due to space limitation, in all tables in this section, we only list the information regarding the configuration settings for the latest version. In the tables presented in this section, Elastic refers to ElasticSearch.

Each project in Table II is highly configurable, and ships with hundreds of settings. The table also presents the number of settings of each type (e.g., boolean, integer, etc.) for each software for the latest version. The column titled "Other" is used to list the number of settings that use complex data structure. The rate of false negative for our tool is shown in Table III in which we compared the extracted list of configuration settings of a software package against the list of configuration settings retrieved from the software's official documentation. As can be seen, even for Hadoop, that has a total of 847 documented settings, our tool missed only 4.3% of them, and missed none in case of Cassandra.

Table IV shows the number of newly added, removed, and modified configuration settings for each software in order of version release (e.g., Cassandra version 1.2.19 vs. version 2.0.7, version 2.0.7 vs. version 2.1.8). As can be seen, configuration settings are changed in every new version for each of these software.

The rate of false negative for change detection for different software is listed in Table V. As can be seen, CSMiner fails to identify less than 10% of the newly added settings. The rate of false negative for deleted settings is less than 10% for all cases. The reasons behind these false negative cases will be explained in Section V. The time taken by the tool to analyze these software packages is listed in Table VI. As expected, analyzing Hadoop takes longest due to its large size. The size of the constructed CCGs is compared against the complete graph for each program and is presented in Table VII. As can be seen, the size of the CCG is significantly smaller compared to the complete call graph for the whole program.

Table VIII shows the runtime of the graph comparison algorithm that identifies the changes in configuration settings across two different versions (i.e., time for generating Table IV).

### A. Lessons Learned

*1) The reasons behind the configuration change varies widely:* In our study, we observed that a setting is added or removed not always to add/remove a software feature, but sometimes to fix bugs, or to improve the reliability and performance of a software. For example, when a new module was added into ElasticSearch 1.7.0, a setting named "index.unassigned.node_left.delayed_timeout" was introduced to allow users to tweak the performance of this module[10]. In Hadoop 2.6.0, settings "fs.s3a.threads.max", "fs.s3a.threads.core", "fs.s3a.threads.keepalivetime", and "fs.s3a.max.total.tasks" were introduced to avoid OutOfMemoryError bug in S3A FileSystem[11], which occurs in the earlier versions. In HBase version 1.1, users can use the new setting "hbase.cells.scanned.per.heartbeat.check" to control the performance of the scan operation (i.e., a step in a querying process)[12].

*2) Changes in configuration settings are not always reflected in the code:* We observed that, even after a configuration setting is removed, the code related to that setting may not be removed in the new version of the software. Additionally, while some functions that read deleted settings are marked as deprecated, some are not. For example, function *getIndexInterval()* in Cassandra 2.1.8 that reads setting "index_interval" is marked as deprecated while function *getCommitLogPeriodicQueueSize()* that loads the deleted setting "commitlog_periodic_queue_size" is not. Such inconsistency can lead to confusion and erroneous use of settings.

*3) Changes in software due to changes in settings can be hard to understand and follow:* From our study, we conclude that, without an automated tool, it is non-trivial to keep track of the changes in code base due to the frequent addition, removal, and/or modification to classes that use configuration settings. For example, class *ScheduledRangeTransferExecutorService*, which uses the setting "num_tokens", is added in Cassandra version 2.0.7, but later removed in version 2.1.8.

[10]https://www.elastic.co/guide/en/elasticsearch/reference/2.0/delayed-allocation.html

[11]https://wiki.apache.org/hadoop/AmazonS3

[12]https://blogs.apache.org/hbase/entry/scan_improvements_in_hbase_1

| Program | Cassandra (1.2-2.0) | Cassandra. (2.0-2.1) | Elastic (1.5-1.7) | Elastic (1.7-2.1) | Hadoop (2.2-2.6) | Hadoop (2.6-2.7) | HBase (0.90-0.98) | HBase (0.98-1.1) |
|---|---|---|---|---|---|---|---|---|
| No. added settings | 8 | 25 | 33 | 21 | 118 | 86 | 98 | 19 |
| No. removed settings | 8 | 8 | 22 | 68 | 19 | 19 | 8 | 10 |
| No. modified settings | 25 | 12 | 15 | 22 | 24 | 14 | 26 | 17 |

Table IV: The summary of configuration setting changes across multiple versions.

| Program | Rate of False Negative for New Settings (%) | Rate of False Negative for Removed Settings (%) |
|---|---|---|
| Cassandra (1.2-2.0) | 0 | 0 |
| Cassandra (2.0-2.1) | 0 | 0 |
| Elastic (1.5-1.7) | 5.7 | 0 |
| Elastic (1.7-2.1) | 4.5 | 2.8 |
| Hadoop (2.2-2.6) | 5.6 | 0 |
| Hadoop (2.6-2.7) | 7.5 | 9.5 |
| HBase (0.90-0.98) | 2.9 | 0 |
| HBase (0.98-1.1) | 5.0 | 9.0 |

Table V: The percentage of changes the tool missed.

| Program | Execution time (minute) |
|---|---|
| Cassandra | 25 |
| Elastic | 29 |
| Hadoop | 52 |
| HBase | 38 |

Table VI: Runtime of the tool.

| Program | Size of the constructed CCG | Size of the complete graph |
|---|---|---|
| Cassandra | 3,253 | 151,260 |
| Elastic | 1,116 | 93,674 |
| Hadoop | 6,718 | 823,002 |
| HBase | 5,380 | 521,168 |

Table VII: Size of the constructed CCG vs. size of the complete graphs.

| Program | Execution time (sec) |
|---|---|
| Cassandra | 210 |
| Elastic | 210 |
| Hadoop | 240 |
| HBase | 225 |

Table VIII: Time to populate Table IV.

Because of this, the usage scenario of setting "num_tokens" in version 1.2.19 and 2.1.8 is different compared to version 2.0.7. In another example, the setting "partitioner" is used by a class named *AntiEntropyService* in Cassandra version 1.2.19, then the name of the class was changed to *Validator* in Cassandra version 2.0.7. In Cassandra version 2.1.8, the class *Validator* still exists but the setting "partitioner" has been removed from the class. While the underlying reasons behind such inconsistency could not be confirmed, our tool can automatically identify such evolution across multiple versions, making system administrators aware of potential issues while upgrading to a different version.

*4) Modification in configuration settings dominates the type of change:* From our study, we observed that the change in the way an existing setting is used and/or loaded outnumbers the number of addition/deletion of settings. The number of such changes in our study is listed in Table X. We categorize such changes into three groups as follows: if there is a change in the CCG for a setting, that is listed under row "CCG", if there is a change in the CDF for a setting, that is listed under row "CDF", and if for a setting both CCG and CDF is changed in the new version, that is listed under row "CCG and CDF". For example, if a new code branch is added in the new version that leads to a configuration setting, that is considered a change in CCG. If a code fragment is added in the new version of the program that uses a configuration setting as a predicate variable, that is considered a change in the CDF.

In Table X we can see that most of the changes occur in CDF, which implies that the same configuration setting is used in different ways in the new version. For example, in HBase, the setting "hbase.regionserver.global.memstore.upperLimit" is used by class *MemStoreFlusher* in version 0.90, class *HBaseConfiguration* and *MemStoreFlusher* in version 0.98, and class *HeapMemorySizeUtil* in version 1.1. This finding underscores the need for an automated tool that can assist administrators/developers in identifying and understanding the changes in configuration settings across multiple versions.

Finally, we also looked into whether the type of a configuration setting is changed across versions or not (e.g., change to long from integer). However, we could not find any such cases in our case studies.

*5) Changes in configuration settings are not always reflected in the documentation:* Finally, in this work, we identified that developers often do not document the changed settings. The number of changed configuration settings which are not documented for each software is listed in Table XI. This finding of our is consistent with prior work that also reported the lack of documentation for configuration settings [21]. In addition to inconsistent documentation, a number of newly added configuration settings are not described in the change logs or release notes of new versions either. The changes made to configuration settings are rarely reported and we cannot find any website which lists the changed settings after a new version of a software is released. The lack of documentation of configuration settings is described in the Apache Hadoop issue tracker log as well: "*added dfs.web.authentication.filter but this doesn't appear to be documented anywhere.*"[13] We also noted that

[13]https://issues.apache.org/jira/browse/HDFS-7033

| Program | Cassandra (1.2-2.0) | Cassandra. (2.0-2.1) | Elastic (1.5-1.7) | Elastic (1.7-2.1) | Hadoop (2.2-2.6) | Hadoop (2.6-2.7) | HBase (0.90-0.98) | HBase (0.98-1.1) |
|---|---|---|---|---|---|---|---|---|
| No. added settings | 8 | 25 | 35 | 22 | 125 | 93 | 101 | 20 |
| No. removed settings | 8 | 8 | 22 | 70 | 19 | 21 | 8 | 11 |

Table IX: The summary of configuration setting changes extracted from documented settings across multiple versions.

| Program | Cassandra (1.2-2.0) | Cassandra. (2.0-2.1) | Elastic (1.5-1.7) | Elastic (1.7-2.1) | Hadoop (2.2-2.6) | Hadoop (2.6-2.7) | HBase (0.90-0.98) | HBase (0.98-1.1) |
|---|---|---|---|---|---|---|---|---|
| CCG | 6 | 3 | 4 | 8 | 8 | 3 | 7 | 4 |
| CDF | 12 | 6 | 5 | 8 | 11 | 6 | 12 | 10 |
| CCG and CDF | 7 | 3 | 6 | 6 | 5 | 5 | 7 | 3 |

Table X: Type of configuration setting modifications across multiple versions.

a setting might be introduced in an early version but is only documented in a later version of the software. For example, while the setting "dfs.datanode.scan.period.hours" was added in Hadoop version 2.2 and earlier versions, the description of that setting is added in Hadoop version 2.7. Please also note that, as the modification of configuration settings are often not documented, developers often need to manually search the property file of the software, the git commit log, or the Q&A sites such as Stack Overflow to find the changes. However, users can only search for the information regarding a setting only if they know its name, which is not available to begin with, which is addressed by our tool as well that reveals the newly added settings automatically.

| Program | Number of undocumented settings |
|---|---|
| Cassandra | 4 |
| Elastic | 3 |
| Hadoop | 5 |
| HBase | 3 |

Table XI: The number of undocumented changed configuration setting.

## V. Discussion and Threats to Validity

While our tool was able to identify the changes in configuration settings across multiple versions with high accuracy, in this section, we would like to highlight the key limitations of our work as follows.

*First,* due to the nature of large scale software (e.g., use of third party libraries), it is possible that the CoG may not be complete, and miss a small number of cases. Specifically, as the accuracy of the CoG depends on the accuracy of the list of starting points that is used to construct the call graph, if the starting point that leads to the configuration loading point is not reachable, it can lead to an incomplete CoG. To address this (to some extent), we turned on the setting "all-reachable" of Soot tool so it considers all methods of a program to be reachable and includes them in the call graph construction phase. However, as many starting points are invoked externally from client side in system software like Cassandra or Hadoop, this may not work in those cases. For example, a function can be invoked via RPC protocol (e.g., Apache Thrift [25]), or called directly by using public APIs. Although we have created fake starting points to invoke all RPC protocol methods to address this issue, our approach may still miss load points of some configuration settings. This is one of the main reasons which causes false negative cases as described earlier. However, given that the number of such special cases is small (based on our experience), we strongly believe that our tool will still cover most of the cases, which is demonstrated by our thorough evaluation. Furthermore, due to the possibility of incomplete documentation, it was not possible to claim identified settings that are not listed in the official documentation as false positive in our study. Therefore, in this paper we only present the rate of false negatives to show what percentage of documented configuration settings the tool can successfully identify automatically. Please note that, while our tool can identify the settings and changes across multiple versions effectively, it cannot make any recommendation regarding the optimal combination of settings, which is a different research problem.

*Finally,* while we studied four large-scale open source projects, all the findings may not be equally applicable for all open source projects. However, we strongly believe that widely used open source software such as Hadoop and Cassandra with dozens of active developers are representative of large-scale open source software, making our findings compelling.

## VI. Conclusion

This paper presents CSMiner, a tool to identify changes of configuration settings across multiple versions of large scale software, and introduces the concept of configuration-oriented graph that helps users answering questions such as *"Where and How setting X is used in my program?"*, *"Which configuration settings have been added or removed in the new version?"*, and *"Which configuration settings have been*

*modified in the new version and how?"*. Extensive evaluation of the tool using four different open source software packages is presented to demonstrate the effectiveness of the presented approach.

## VII. Acknowledgement

This work is supported by the AFOSR under Grant No. FA 9550-15-1-0184. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency.

## References

[1] 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*. New York, NY, USA: ACM, 2011.

[2] S. Zhang and M. D. Ernst, "Which Configuration Option Should I Change?" in *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 152–163.

[3] T. C. Lethbridge, J. Singer, and A. Forward, "How Software Engineers Use Documentation: The State of the Practice," *IEEE Softw.*, vol. 20, no. 6, pp. 35–39, Nov. 2003.

[4] J. Singer, "Practices of Software Maintenance," in *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1998.

[5] "Soot: A Framework for Analyzing and Transforming Java and Android Applications," http://sable.github.io/soot/.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014.

[7] "Apache Cassandra," http://cassandra.apache.org/.

[8] "Elasticsearch," https://www.elastic.co/products/elasticsearch.

[9] "Apache Hadoop," http://hadoop.apache.org/core/.

[10] "Apache HBase," http://hbase.apache.org/.

[11] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating Range Fixes for Software Configuration," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 58–68.

[12] 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*. Berkeley, CA, USA: USENIX Association, 2011, pp. 28–28.

[13] A. Rabkin and R. Katz, "Precomputing Possible Configuration Error Diagnoses," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 193–202.

[14] S. Zhang and M. D. Ernst, "Automated Diagnosis of Software Configuration Errors," in *Proceedings of International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 312–321.

[15] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software," in *Proceedings of the 10th OSDI*. Berkeley, CA, USA: USENIX Association, 2012, pp. 307–320.

[16] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic Misconfiguration Troubleshooting with Peerpressure," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004.

[17] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration Debugging As Search: Finding the Needle in the Haystack," in *Proceedings of the 6th OSDI*. Berkeley, CA, USA: USENIX Association, 2004.

[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 ASE*. New York, NY, USA: ACM, 2014.

[19] N. Nguyen and M. M. H. Khan, "Performance Analysis of a Fault-tolerant Exact Motif Mining Algorithm on the Cloud," in *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*, Dec 2013.

[20] M. Lillack, C. Kästner, and E. Bodden, "Tracking Load-time Configuration Options," in *Proceedings of the 29th International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2014.

[21] A. Rabkin and R. Katz, "Static Extraction of Program Configuration Options," in *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011.

[22] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61.

[23] E. Bodden, "Inter-procedural Data-flow Analysis with IFDS/IDE and Soot," in *Proceedings of the International Workshop on State of the Art in Java Program Analysis*. New York, NY, USA: ACM, 2012.

[24] "Apache JMeter," http://jmeter.apache.org/.

[25] "Apache Thrift," https://thrift.apache.org/.