ConfEx: Towards Automating Software Configuration Analytics in the Cloud

Ozan Tuncer*, Nilton Bila†, Sastry Duri†, Canturk Isci†, and Ayse K. Coskun*
* Boston University, {otuncer, acoskun}@bu.edu
† IBM Research, {nilton, sastry, canturk}@us.ibm.com

***Abstract*—Modern cloud applications are designed in a highly configurable way to ensure increased reusability and portability. With the growing complexity of these applications, configuration errors (i.e., misconfigurations) have become major sources of service outages and disruptions. While some research has so far focused on detecting errors in configurations that are represented as well-structured key-value pairs, the configurations of cloud applications are typically stored in text files with application-specific syntax and in unlabeled file system locations, limiting the use of existing error detection tools.**

This paper introduces ConfEx, a framework that enables *discovery and extraction* of text-based configurations in multi-tenant cloud platforms and cloud image repositories for configuration analysis and validation. ConfEx uses a novel vocabulary-based technique to identify text-based configuration files in cloud system instances with unlabeled content, and leverages existing configuration parsers to extract the information in these files. We show that ConfEx achieves over 98% precision and recall in identifying configuration files on 3893 popular Docker Hub images and we also demonstrate a use case of ConfEx for detecting injected misconfigurations via outlier analysis.

I. Introduction

Cloud applications have complex architectures, often comprising many software components adopted from the open source or a variety of vendors that must work in tandem. To function correctly, securely, and with high performance, these applications often depend on precise tuning of hundreds of configuration parameters. As application architectures grow more complex, developers no longer hold full mastery of every configuration knob used by each software component in their applications. This complexity and resulting knowledge gap has elevated software configuration errors to become a leading cause of cloud software failures [1]. The problem is exacerbated because applications seldom validate their configurations before using them. Recent work has shown that, depending on the application, the software lacks any special error checking code to validate 14% to 93% of its configuration parameters [2].

The research community has developed various tools to automatically check for errors in an application-agnostic manner (e.g., [3], [4]). Among such tools, statistical and learning-based techniques (e.g., [5], [6]) have gained popularity as low overhead configuration checkers. Statistical configuration checkers train on a corpus of configurations and learn common patterns, and then identify configurations that deviate from the norm as potential errors.

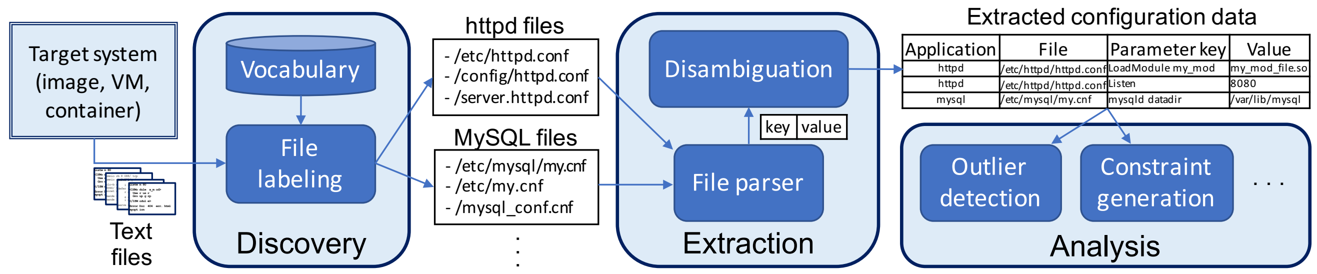
Training learning-based models and using them for validation of configurations requires *discovery* of configurations and *extraction* of configuration parameters across large populations of installed applications. However, our experience with cloud applications has revealed that configurations are often stored in non-standard locations in the file system, which complicates the task of configuration discovery. Moreover, the configuration parameters are often embedded in human readable text files that are difficult to extract reliably with automated tools outside of the native software that is co-developed with the configuration format. For effective application-agnostic analysis, the information extracted from these files needs to be represented in a *consistent* format that allows comparison of individual parameters across different files.

This paper introduces *ConfEx*, a novel software configuration analytics framework that enables robust analysis of text-based configurations in multi-tenant cloud platforms and image repositories. ConfEx discovers configuration files of known applications in cloud instances (i.e., images, VMs, and containers) and parses these files to produce consistent configuration data for corpus-based analysis. We demonstrate a use case of ConfEx on a corpus of 3893 popular Docker Hub images by detecting injected misconfigurations through outlier analysis. Our contributions can be summarized as follows:

- We present our design and implementation of ConfEx, a *configuration analytics framework* that enables *discovery and extraction* of consistent configuration data and robust configuration analysis in multi-tenant cloud platforms.
- As part of our framework, we develop a vocabulary-based *configuration file discovery* technique to identify text-based software configuration files in cloud instances with unlabeled content. Our approach identifies application configuration files with over 98% precision and recall.
- We show that the outputs of existing configuration file parsers often lack the consistency and robustness needed for statistical analysis, and design a *disambiguation* methodology on parser outputs to resolve this problem. The impact of disambiguation is especially visible when performing outlier analysis to detect misconfigurations.

II. ConfEx Configuration Analytics Framework

Figure 1 shows an overview of ConfEx. ConfEx first *discovers* the configurations files in cloud system instances with unlabeled content using an offline-generated vocabulary. It then extracts the configuration data from these files using a



Application	File	Parameter key	Value
httpd	/etc/httpd/httpd.conf	LoadModule my_mod	my_mod_file.so
httpd	/etc/httpd/httpd.conf	Listen	8080
mysql	/etc/mysql/my.cnf	mysqld datadir	/var/lib/mysql

Fig. 1. ConfEx overview. Given a target system instance, *File labeling* examines the contents of text files and labels configuration files with the name of the software they belong to. Based on this label, the *File parser* extracts the file content and produces key-value pairs using software-specific parsing rules. The *Disambiguation* step transforms the parser output into consistent key-value pairs, where a key corresponds to a single configuration parameter consistently across different system instances. The extracted configuration data is then used for corpus-based analysis such as outlier detection and constraint generation.

community-driven configuration parser and applies a software-specific *disambiguation* methodology to prepare the extracted data for analysis. The *extraction* phase generates key-value pairs that represent configuration data. Finally, these key-value pairs are augmented with the software label and the source file path to enable robust corpus-based configuration analysis. The rest of this section described ConfEx's phases in detail.

A. Discovery

We focus on text file based configurations, which are prevalent for many of the building blocks of cloud applications (e.g., MySQL, Nginx, and Redis). Our configuration file discovery technique is based on our observation that parameter names and configuration commands, which we refer to as *important words*, can be used to associate text files with applications. Figure 2 depicts *ConfEx*'s file discovery step in detail. During offline training, we use known configuration files that are labeled with application names. We then generate application-specific vocabularies by extracting the *important words* in each of these files in an application-agnostic way as follows: We first discard commented-out lines in a file as comments typically contain descriptions of the configuration options with few (or no) application-specific words. We focus on the first word in the remaining lines as the first words of non-comment lines in a configuration file typically correspond to parameter names or configuration commands, whereas the subsequent words are user-provided values such as integers and file paths. While extracting the first word of a line, we use the following characters as delimiters to account for the characters that are commonly used as part of a configuration file syntax: `\t`, `=`, ` `, `:`, `<`, `>`, `[`, `]`, `,`.

During testing, we use the same methodology to extract the set of *important words* in an input text file. We then calculate the similarity of the input *important word* set to each *important word* set in the vocabulary of each application. To calculate the similarity of a set pair, we use the Jaccard index, defined as $|A \cap B|/|A \cup B|$, where A and B are two sets. If the maximum achieved similarity using an application vocabulary is larger than a certain threshold, $T_{confidence}$, the file is labeled as a candidate configuration file of that application.

To reduce the overhead of ConfEx, we only perform analysis on text files that are smaller than 200KB. This threshold is supported by our investigation of 3893 Docker Hub images on which the largest configuration file found was 36KB.

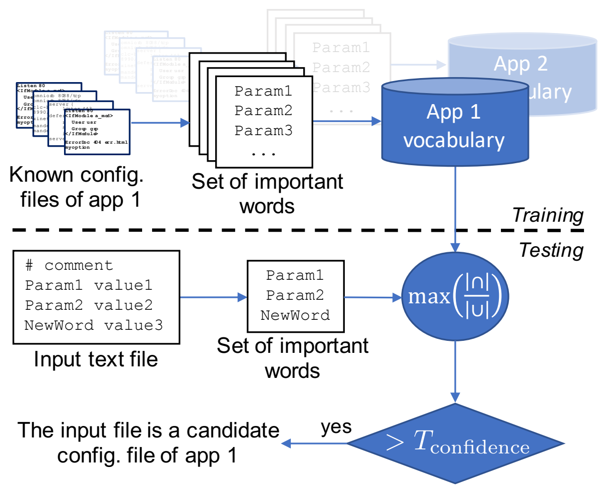


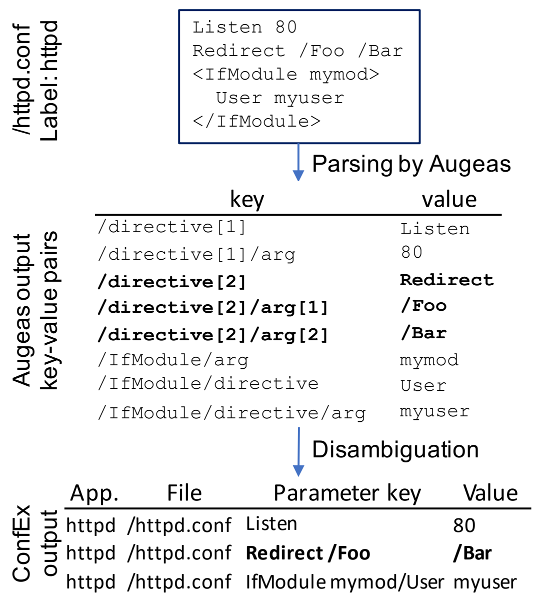
Fig. 2. Discovery phase. During offline training, a vocabulary is generated for each application using known configuration files. Input text files are compared with each application vocabulary and selected as candidate configuration files upon a match that is larger than a confidence threshold.

B. Extraction

While existing studies on configuration analysis have mostly focused on configuration stores that do not require data extraction such as Windows Registry (e.g., [7]), or configurations with standard file formats such as `XML` and `JSON` (e.g., [4], [8]), most configuration files in today's cloud services (such as httpd and Nginx) are kept in human-readable text files that do not use standard file formats. The variety and rapid evolution of applications make it expensive and bug-prone to implement and maintain custom parsers for different applications for every configuration analysis tool.

To leverage the knowledge of domain experts on various applications and re-use an existing code-base that is continuously maintained, we build our extraction phase on top of Augeas [9], which is one of the most popular tools available today for automatized configuration parsing and editing. Augeas has extensive software configuration coverage including common cloud applications such as httpd, MySQL, Nginx, PHP, and PostgreSQL.

As Augeas is primarily intended for managing configurations in systems with uniform and known configuration structure, its output is not ideal for key-value-based analysis. As seen in Fig. 3, Augeas produces artificial keys (e.g., `/directive[1]`) that do not correspond to parameters but



```
Listen 80
Redirect /Foo /Bar
<IfModule mymod>
  User myuser
</IfModule>
```

```
key                          value
/directive[1]                Listen
/directive[1]/arg            80
/directive[2]                Redirect
/directive[2]/arg[1]         /Foo
/directive[2]/arg[2]         /Bar
/IfModule/arg                mymod
/IfModule/directive          User
/IfModule/directive/arg      myuser
```

App.	File	Parameter key	Value
httpd	/httpd.conf	Listen	80
httpd	/httpd.conf	**Redirect /Foo**	**/Bar**
httpd	/httpd.conf	IfModule mymod/User	myuser

Fig. 3. Extraction phase. Augeas parses configuration files based on the labels given by the discovery phase and generates key-value pairs that represent configurations. Then, using software-specific rules, Augeas' output is transformed into a format where a key corresponds to a single parameter consistently across different files.

represent the location and type of the configuration entries. Hence, a specific Augeas key does not necessarily point to the same parameter across different files. To resolve this issue, we write software-specific transformation rules and transform Augeas' output into a standardized format shown in Fig. 3, where a key consistently points to the same configuration parameter across different files. We write these rules manually based on an investigation of Augeas' output and applications' documentation, and transform the Augeas output such that the transformed key-value pairs faithfully represent configurations. As an example, one such rule for httpd configurations is to use the first argument of a `Redirect` directive in the parameter key and the second argument of it as the value of the parameter (marked with bold text in Fig. 3).

C. Analysis

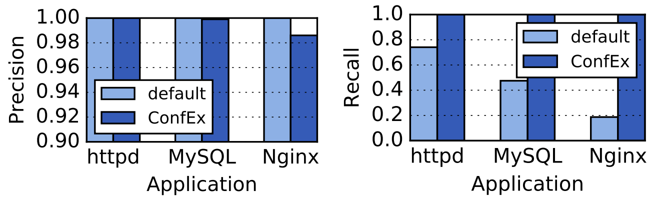
The discovery and extraction phases of ConfEx produces consistent key-value pairs, enabling various configuration analysis techniques in the cloud. A rich variety of analysis methods can be applied as part of ConfEx, including outlier value detection [6] and rule-based validation [5], [10].

III. Evaluation

We focus on the Docker Hub images that contain either the network services system configuration file (`/etc/services`) or one of the three following popular cloud applications: httpd, MySQL, and Nginx. For `/etc/services`, we use the most downloaded thousand images and discard the images that do not have `/etc/services`. For each application, we use the images that contain the application name in their name or description and are downloaded at least 50 times. We have manually labeled the configuration files in these images by examining

TABLE I
Statistics on the studied Docker Hub Images

application	# of images	total # of config. files	total # of text files
httpd	272	9191	330106
MySQL	715	2600	509857
Nginx	2906	22450	313357
Network services	726	726	not used for discovery



(a) precision (b) recall

Fig. 4. Comparison of the default path-based and ConfEx's vocabulary-based discovery approaches ($T_{\text{confidence}} = 0.5$). The default approach can identify only 19-74% of the application configuration files.

file contents and file paths. Table I summarizes the number of images we use in our evaluation along with the number of text files and identified configuration files in these images.

A. Discovery

We compare ConfEx's configuration discovery with a common approach that searches for configuration files only in well known file system locations. We use the file locations checked by Augeas. We measure the effectiveness of discovery separately for each application and using five-fold cross validation, where we repeat each cross validation ten times with different randomly-selected partitions.

We use *precision* and *recall* as evaluation metrics. Precision is the fraction of true positives (i.e., correctly predicted configuration files) to the total number of files predicted as configuration files, and recall is the fraction of true positives to the total number of configuration files in the testing set.

In our experiments, we find that using a similarity threshold larger than 0.5 has negligible impact on precision and recall, and hence, use $T_{\text{confidence}} = 0.5$. Figure 4 compares the default (Augeas) discovery approach with ConfEx's vocabulary-based discovery. As the standard configuration file paths do not contain text files that are not configurations, the default approach achieves ideal precision. However, as shown in Fig. 4b, only 19% of Nginx configuration files can be found with this approach as the remaining configuration files are not located in the default paths in the target images.

Overall, ConfEx successfully identifies 34156 target configuration files (out of 34241), while the baseline can identify only 12249 of the configuration files. In the remaining 85 files that are missed by ConfEx, approximately half of the parameter names are uncommon and are not seen during training. ConfEx's lowest precision is observed with Nginx, where ConfEx mislabels 347 files as Nginx configurations. These mislabeled files contain words that are used as parameter names in Nginx such as the word `include` in file `/etc/ld.so.conf`.

TABLE II
INJECTED APPLICATION MISCONFIGURATIONS

application	name	description
httpd	url	Error 401 points to a remote URL
httpd	dns	Unnecessary reverse DNS lookups
httpd	path	Wrong module path
httpd	mem	MaxMemFree should be in KB
httpd	req	Too low request limit per connection
MySQL	enum	Enumerators should be case-sensitive
MySQL	buf	Unusually large sort buffer
MySQL	limit	Too low connection error limit
MySQL	max	Invalid value for max number of connections
Nginx	files	Too few open files are allowed per worker
Nginx	debug	Logging debug outputs to a file
Nginx	access	Giving access to root directory
Nginx	host	Using hostname in a listen directive

B. Detecting Misconfigurations via Outlier Analysis

We present a use case of ConfEx by automatically detecting injected misconfigurations using PeerPressure [6]. PeerPressure is originally designed for Windows registry, and finds the culprit configuration entry in an image with a single configuration error, where configurations are provided as key-value pairs. It ranks the configurations based on their probability of being an error, which is calculated using outlier analysis via Bayesian estimation. We apply PeerPressure on all files discovered by ConfEx. We also show the impact of ConfEx's disambiguation step on PeerPressure's accuracy by using the default Augeas output before disambiguation as a baseline.

We use PeerPressure to detect the application misconfigurations listed in Table II as well as synthetic `/etc/services` misconfigurations. For applications, we inject each misconfiguration listed in Table II to a randomly selected image that contains the target parameter to be misconfigured. To generate `/etc/services` misconfigurations, we randomly select a service in a randomly chosen image, and change the port used by the selected service to a random integer between 1 and 10000. For each target misconfiguration, we repeat the randomized injection 1000 times.

Figure 5 shows the percentage of injected errors that are ranked within the top five suspects by PeerPressure among 1000 randomized injections of our target misconfigurations. Using ConfEx's disambiguated keys consistently leads to similar or higher rankings compared to using default Augeas keys, making it easier to pinpoint the injected error.

With the default keys, PeerPressure suffers from having an incorrect view on the distribution of configurations. This problem becomes more significant when the number of keys that represent the misconfigured parameter across the corpus is large (e.g., more than five), such as in `services` misconfigurations. Moreover, when the misconfigured image has files that have substantially different parameter ordering compared to the files seen in the corpus, common configuration entries use uncommon keys and become outliers in the corpus, resulting in high error probability. However, PeerPressure can still detect the injected errors with the default keys if the parameter

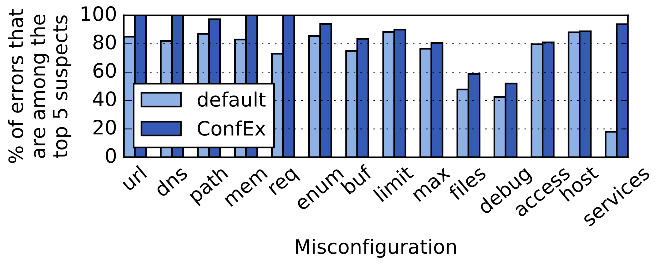


Fig. 5. The percentage of injected errors that are ranked within the top five suspects by PeerPressure among 1000 randomized injections for each misconfiguration. `services` is the `/etc/services` misconfiguration.

ordering in the misconfigured image is similar to the majority of images seen in the corpus.

In *files* and *debug* errors, outlier detection performs poorly both with the default Augeas keys and ConfEx's disambiguated keys. This is because compared to the other injected errors, the parameters being misconfigured in *files* and *debug* have fewer instances in the corpus. Hence, the injected erroneous values are not perceived as an outlier by PeerPressure. This can be avoided by using a larger configuration corpus.

IV. CONCLUSION

In this work, we have introduced ConfEx, a framework to discover and analyze text-based software configurations in multi-tenant cloud platforms. Our framework enables the use of existing configuration analysis tools, which are designed for key-value pairs, with text file based configurations in the cloud. Our results have shown that ConfEx achieves over 98% precision and recall on identifying configuration files, and our disambiguation method consistently improves the efficacy of detecting configuration errors through outlier analysis.

REFERENCES

[1] Z. Yin *et al.*, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 159–172.

[2] T. Xu *et al.*, "Early detection of configuration errors to reduce failure damage," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[3] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *OSDI*, 2012, pp. 307–320.

[4] F. Behrang, M. B. Cohen, and A. Orso, "Users beware: Preference inconsistencies ahead," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 295–306.

[5] J. Zhang *et al.*, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 687–700.

[6] H. J. Wang *et al.*, "Automatic misconfiguration troubleshooting with peerpressure," in *OSDI*, 2004, pp. 17–17.

[7] D. Yuan *et al.*, "Context-based online configuration-error detection," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011, pp. 28–28.

[8] S. Zhang and M. D. Ernst, "Proactive detection of inadequate diagnostic messages for software configuration errors," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 12–23.

[9] D. Lutterkort, "Augeas–a configuration api," in *Linux Symposium*, 2008, pp. 47–56.

[10] S. Baset *et al.*, "Usable declarative configuration specification and validation for applications, systems, and cloud," in *Proceedings of the Industrial Track of the 18th International Middleware Conference*, 2017.