RC25675 (WAT1803-107) March 29, 2018
Computer Science

IBM Research Report

ConfEx: An Analytics Framework for Text-Based Software Configurations in the Cloud

Ozan Tuncer[1], Nilton Bila[2], Canturk Isci[2], Ayse K. Coskun[1]

[1]Boston University
Boston, MA 02215 USA

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

ConfEx: An Analytics Framework for Text-based Software Configurations in the Cloud

Ozan Tuncer
Boston University
Boston, MA 02215
Email: otuncer@bu.edu

Nilton Bila
IBM Research
Yorktown Heights, NY 10598
Email: nilton@us.ibm.com

Canturk Isci
IBM Research
Yorktown Heights, NY 10598
Email: canturk@us.ibm.com

Ayse K. Coskun
Boston University
Boston, MA 02215
Email: acoskun@bu.edu

***Abstract*—Modern cloud applications are designed in a highly configurable way to provide 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 automatically detecting errors on configurations that are represented as well-structured key-value pairs, *discovering and extracting* configurations remain a challenge for a wide range of cloud applications that store their configurations in loosely-structured text files.**

This paper proposes ConfEx, a framework that enables discovery and analysis of text-based configurations in multi-tenant cloud platforms and cloud image repositories. Our framework uses a novel vocabulary-based discovery technique to identify text-based configuration files in cloud system instances with unlabeled content. We show that, even for labeled configuration files, widely-used and expert-maintained configuration parsing tools lack the consistency and robustness needed for meaningful statistical analysis of configurations. We introduce a novel disambiguation technique that resolves the inconsistencies in the configuration-related data extracted by existing parsers. When tested on 4581 popular Docker Hub images, ConfEx achieves over 98% precision and recall in identifying configuration files, and consistently improves the efficacy of misconfiguration detection through outlier analysis as well as syntactic configuration validation.

I. Introduction

Cloud software is complex and highly customizable. To function correctly, securely, and with high performance, cloud applications often depend on precise tuning of hundreds of configuration parameters [1]. In typical cloud services that consist of multi-tiered software stacks, ensuring the desired operation often requires correctly configuring thousands of parameters [2].

Errors in software configurations have been reported as causes of service disruptions and outages at Facebook [3], LinkedIn [4], Microsoft Azure [5], Amazon EC2 [6], and Google [7]. Moreover, the affordability offered by the cloud and the prevalence of open-source software have enabled new levels of agility, where small teams of developers can deliver new cloud services and functionality in short periods of time. This newfound agility has led to a trend where service developers and operators may lack the expertise needed to precisely tune all software components of a multi-tier architecture. As a result, misconfigurations have become one of the lead causes of cloud software failures [8], [9], [10].

Configurations are traditionally validated by applications during startup. However, recent work has shown that 14-93% of configuration parameters in today's cloud software do not have any special code for checking their correctness during application initialization [11]. To detect misconfigurations before deployment, researchers have developed various tools to automatically check for errors in application configurations (e.g., [12], [13]). Among such tools, statistical and learning-based techniques (e.g., [14], [15], [16]) have gained popularity as low overhead configuration checkers that can be applied in an application-agnostic manner. Statistical configuration checkers train on a corpus of configurations and learn common patterns. These methods can then identify configurations that deviate from the norm as potential errors. Such statistical methods are powerful in practice because they do not require intrusive static/dynamic analysis or application instrumentation.

In order to perform statistical and learning-based configuration analysis in multi-tenant cloud platforms, it is essential to extract configuration information from cloud system instances (i.e., images, VMs, and containers) without losing any information that is crucial for detecting errors. This is challenging because cloud instance contents are largely unlabeled. One needs to discover which files are configuration files and also figure out to which applications these files belong. Furthermore, cloud software configurations are typically stored in loosely-structured text files where each software has its own custom configuration syntax. For effective statistical analysis, the information extracted from these files needs to be represented in a consistent format that allows comparison of individual configuration parameters across a large number of cloud instances.

In this work, we propose *ConfEx*, a novel software configuration analytics framework that enables robust analysis of loosely-structured text-based configurations in multi-tenant cloud platforms and image repositories. ConfEx discovers configuration files of known applications in cloud instances and parses these files to produce consistent configuration data for corpus-based analysis. We demonstrate two use cases of ConfEx on a corpus of 4581 popular Docker Hub images: (1) detecting injected misconfigurations through outlier analysis and (2) syntactic configuration validation. Our contributions can be summarized as follows:

- We design and implement ConfEx, a *configuration an-*

alytics framework that enables discovery and extraction of consistent configuration data and robust configuration analysis in multi-tenant cloud platforms. We demonstrate that ConfEx enables the use of existing configuration analysis tools, which are designed for key-value pairs, with text-based software configurations in the cloud.

- 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 can identify 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 introduce a *disambiguation technique* for parser outputs to resolve this problem.

The rest of this work starts with an overview of configuration analysis and management techniques in the cloud. Sec. II provides a background on configuration files and common misconfigurations. Section III gives the details of our proposed ConfEx framework. Section IV explains our experimental methodology, and Section V presents our experimental findings. Finally, we conclude in Section VII.

II. Background on Text-based Configurations

In this section, we explain how cloud applications and services typically store their configurations. Then, we categorize common configuration errors to give some insight on the type of information required for effective configuration analysis.

A. Text-based Configurations

Most cloud applications and system services store their configurations in human-readable text files or in configuration stores such as `etcd` and Windows registry. We focus on text file based configurations as this type of storage is prevalent for many of the building blocks of cloud applications (e.g., MySQL, Nginx, and Redis).

Figure 1 shows a snippet from an Apache HTTP server (httpd) configuration file. Each of the first two lines contains a *parameter* followed by a *value*, separated by a space. Lines 3-6 are in an application-specific format representing a conditional

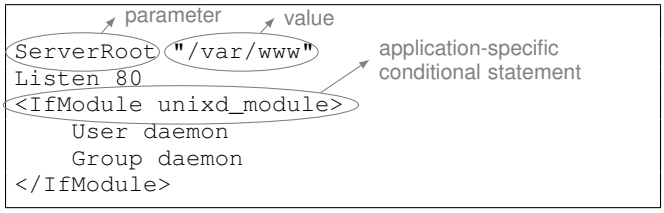


Fig. 1. Httpd configuration file snippet. Configurations are stored in an XML-like format.

```
1 proc   swap     swap   pri=42          0 0
2 tmpfs  /dev/shm tmpfs  mode=0777       0 0
3 devpts /dev/pts devpts defaults,gid=5 0 0
```

Fig. 2. `/etc/fstab` snippet. Configurations are stored in a table format where certain table cells contain multiple configuration entries.

TABLE I
Common configuration error types and example constraints that lead to errors upon violation.

Error type	Example configuration constraint
Illegal entries	In PostgreSQL, parameter values that are not simple identifiers or numbers must be single-quoted.
	Variables must be in certain types (e.g., float).
Inconsistent entries	In PHP, `mysql.max_persistent` must be no larger than the `max_connections` in MySQL.
	In Cloudshare, service's `redis.host` entry (an IP address) must be a substring of Nginx's `upstream.msg.server` entry (IP address:port).
Invalid ordering	When using PHP in Apache, `recode.so` must be defined before `mysql.so`.
Environmental inconsistency	In MySQL, maximum allowed table size must be smaller than the memory available in the system
	In httpd, Apache user permissions must be set correctly to enable file uploads for website visitors.
Missing parameter	In OpenLDAP, a configuration entry must include `ppolicy.schema` to enable password policy.
Valid entries that cause performance or security issues	MySQL's `Autocommit` parameter must be set to `False` to avoid poor performance under "insert" intensive workloads.
	Debug-level logging must be disabled to avoid performance degradation.

statement. While parsing these lines, one needs to retain the relational information between the parameters defined within the conditional statement, indicating that `User` and `Group` belong to the `IfModule unixd_module` section.

In some configuration files, the file schema is not embedded in the file itself and requires domain knowledge to understand. One such example is the Linux filesystem configuration file (`/etc/fstab`), which defines available filesystems and their mount options. As shown in Figure 2, this file is structured in a table format where some columns may include parameter-value pairs such as `pri=42` (line 1) as well as multiple comma-separated entries such as `defaults,gid=5` (line 3).

Extracting configuration data from text-based files requires expertise on the specific application file format. Hence, to conduct corpus-based configuration analysis on a large number of applications, one should use a community-driven parsing tool that allows contributions of application domain experts.

B. Configuration Errors

A common goal of configuration analysis is detecting configuration errors. Table I summarizes common misconfiguration types we derived from related work (e.g., [2], [10], [17], [18], [19]) and online technical forums (e.g., stackoverflow.com and serverfault.com). *Illegal entries* can be identified through syntactic validation. Detecting *inconsistent entries* and *invalid ordering* requires extracting dependency and correlation information among various parameters. *Environmental inconsistencies* occur when application configurations do not

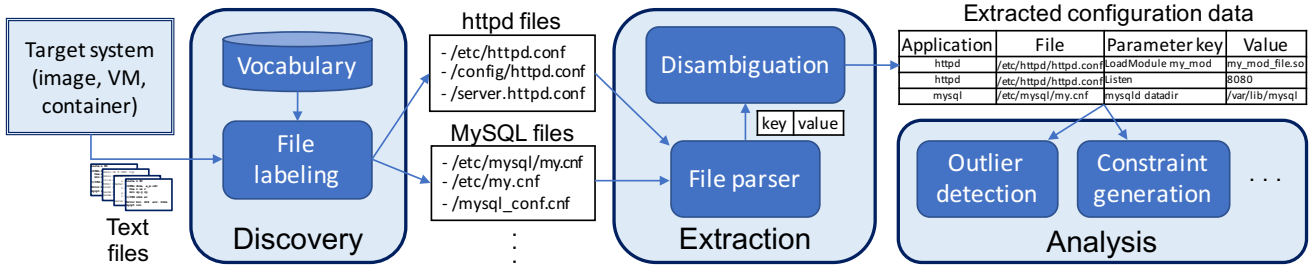


Fig. 3. ConfEx overview. Given a target system instance, *File labeling* examines the contents of text files and labels configuration files with the name of the software they belong to. Based on this label, the *File parser* extracts the file content and produces key-value pairs using software-specific 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.

match the environmental parameters such as file permissions and IP addresses. To find such inconsistencies, one needs to collect and analyze both application and environment configurations. Detecting *missing parameters* requires checking the existence of parameters rather than focusing on the values assigned to parameters. *Valid entries* that cause performance degradation or security vulnerabilities do not lead to crashes or error messages.

Configuration analysis tools commonly treat configurations as key-value pairs, in where each key corresponds to a specific configuration parameter (e.g., [14], [15], [16]). The configuration key-value pairs can be used for detecting the error types shown in Table I except for *invalid ordering*. In this work, we use key-value pairs for configuration analysis and do not focus on *invalid ordering*.

III. ConfEx Configuration Analytics Framework

We propose a configuration analytics framework, ConfEx, for corpus-based configuration analysis in image repositories and multi-tenant cloud platforms. ConfEx discovers the configurations files in cloud system instances with unlabeled content, extracts consistent configuration data from these files, and applies statistical and learning-based analysis methods on the collected data to detect configuration errors.

Figure 3 shows an overview of ConfEx. In the discovery phase, ConfEx uses a vocabulary-based method we designed to discover configuration files. When a new cloud system instance with unlabeled configuration files is introduced (e.g., a new container), ConfEx reads and analyses the text files in the given cloud instance[1] and compares the contents of these files with a vocabulary database that is built offline (details not shown in Fig. 3). When ConfEx discovers a configuration file, it tags the file with a label identifying the software that is associated with the file. These labels are then used in the extraction phase to apply software-specific file parsing and disambiguation rules. The extraction phase generates key-value pairs, which represent configuration data, using keys that consistently correspond to a single configuration parameter across different cloud instances. Finally, these key-value pairs are augmented with the software label and the source file path to enable a comprehensive and robust corpus-based configuration analysis. The rest of this section explains the phases of ConfEx in detail.

[1]ConfEx limits the size of text files inspected to 200KB to maintain low processing overheads. This threshold is supported by our investigation of 4581 Docker Hub images on which the largest configuration file found was 36KB.

A. Discovery

A common approach of locating configuration files is to check specific file system paths based on the locations of standard software installations. While system configuration file locations are typically consistent across different cloud instances, as we show in Section V-A, 26-81% of valid application configuration files are located in non-standard locations in popular Docker Hub images. These files are ignored by the configuration parsing tools. As a result, any configuration problems in these files will not be detected automatically using statistical and learning-based configuration analysis. To resolve this problem, the discovery phase of ConfEx identifies configuration files of known applications in cloud instances in an application-agnostic manner, regardless of where the files are located in the file system.

Figure 4 depicts ConfEx's discovery phase in detail. During offline training, ConfEx reads known configuration files to generate application-specific vocabularies of *important* words, which can be used to associate configuration files with applications. We identify the *important* words based on the following observations:

Observation 1: Commented lines typically contain descriptions of the configuration options with few (or no) application-specific words.

Observation 2: The first word of a non-comment line in a configuration file typically corresponds to a parameter name or a configuration command, whereas the subsequent words in the line are user-provided values such as integers and file paths. Most of such parameter names and configuration commands are specific to an application's configuration.

Observation 3: Certain configuration parameter names and commands (such as `include` and `file`) are used in the configuration files of multiple applications as well as in non-configuration files. Hence, if a file contains lines only starting with such words, this file should not be labeled as

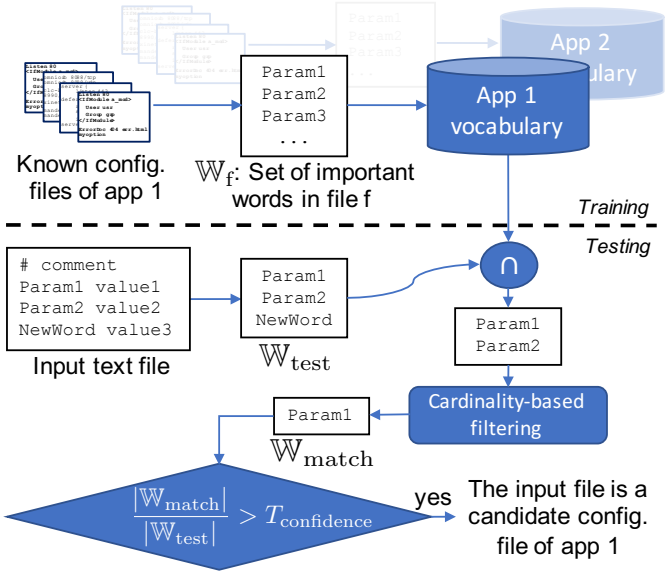


Fig. 4. Discovery phase. A vocabulary is generated for each known application offline. Input text files are compared with each application vocabulary and selected as candidate configuration files upon a match that is larger than a confidence threshold.

a configuration file, unless a similar known configuration file has been previously encountered.

Based on the above observations, we first discard commented lines in the known configuration files that are used for training. We consider a line as a comment if it begins with `//`, `#`, or `%`, excluding the preceding white-space characters (i.e., tab and space). Then, for each file f, we extract the set of unique words ($\mathbb{W}_f$) that appear as the first word in the remaining lines. While extracting the first word from a line, we use for following characters as delimiters to account for the characters that are commonly used as part of a configuration file syntax: `\t`, ` `, `=`, `:`, `<`, `>`, `[`, `]`, `,`. An application vocabulary contains the union of $\mathbb{W}_f$ for all known configuration files of that application.

It is challenging to address *Observation 3* because it is not possible to know all possible words that are used in configuration and non-configuration files during the initial training. To address *Observation 3*, we make use of the cardinality of $\mathbb{W}_f$ ($|\mathbb{W}_f|$) in each known configuration file as an additional similarity metric as follows: For each word w in the vocabulary, we record $min\_cardinality_w$, which is the minimum $|\mathbb{W}_f|$ among all the known configuration files where the word w appears. Then, during testing, we first generate $\mathbb{W}_{\text{test}}$, the set of first words in non-comment lines of the test file, as before. A word w in $\mathbb{W}_{\text{test}}$ is considered as a *match* only if it exists in the vocabulary and if $|\mathbb{W}_{\text{test}}|$ is no less than the recorded $min\_cardinality_w$. In Fig. 4, the set of all such match words in an input file is denoted as $\mathbb{W}_{\text{match}}$.

We use thresholding on the fraction $|\mathbb{W}_{\text{match}}| \,/\, |\mathbb{W}_{\text{test}}|$ to decide whether the input file is a configuration file. If this fraction is larger than a certain $T_{confidence}$ for an application vocabulary, the file is labeled as a candidate configuration file of the corresponding application. As described in detail in Section V-A, we select the confidence threshold empirically and apply the same threshold for all applications.

The syntax of all files that are labeled as configuration files is checked in the extraction phase. If the file does not conform with the configuration file syntax of the target application, users can be warned about a potential syntax error.

To discover the configuration files of a new application, a new application vocabulary should be generated from a set of known configuration files of that application as described above. This vocabulary can be extended simply by processing new labeled files without the need of re-processing the entire set of known configuration files.

B. Extraction

The purpose of the extraction phase is to parse the labeled configuration files and generate key-value pairs that represent configurations. For a robust corpus-based configuration analysis where the input configuration files are curated by different users, the extracted keys should have the following properties:

- *Consistency:* A specific key should always refer to the same parameter, both when observing configurations of a given cloud instance over time, and when comparing configurations across multiple systems.
- *Uniqueness:* Each parameter in a file should be represented by a unique key. However, if two parameters share the same name and context (such as parameters defined as a list), they should share the same key.
- *Context-preserving:* The keys of parameters that appear within the same block of a configuration file must retain this relational information. For example, in Fig. 1, the keys of `User` and `Group` entires must express that both parameters are under the `IfModule` section. Such relations become more prevalent in file formats that keep hierarchical data such as JSON and XML.

While existing studies on configuration analysis have mostly focused on configuration stores that do not require data extraction such as Windows Registry (e.g., [20]), or configurations with standard file formats such as `XML` and `JSON` (e.g., [13], [21]), 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. These files require custom parsing rules based on domain knowledge. However, the variety and rapid evolution of applications make it expensive and bug-prone to implement custom parsers for different applications for every configuration analysis tool.

1) Augeas for Parsing Configuration Files: To leverage the knowledge of domain experts on various applications and reuse an existing code-base that is continuously maintained, we build our extraction phase on top of Augeas [22], which is one of the most popular tools available today for automatized configuration parsing and editing. Augeas has extensive application coverage with 182 *lenses*, which are file parsing rules to generate key-value pairs for different applications including httpd, MySQL, Nginx, PHP, and PostgreSQL. Augeas has been continuously maintained for more than ten years and has interfaces in different programming languages including Python,

Ruby, Perl, and Java. As a result, Augeas is being used by other configuration management tools including Puppet [23] and bcfg2 [24], and also by Encore [17], which is a state-of-the-art configuration analysis tool.

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 statistical analysis and learning in a cloud setting. Several issues with the Augeas parser output can be seen in the example in Fig. 5 and are summarized as follows: The sample input httpd configuration file in Fig. 5 has two `Listen` entries, but these two entries are represented by four key-value pairs after being parsed by Augeas. Similarly, although both `Listen` entries represent the same configuration option, they are referred to using different keys to enforce a unique key per configuration entry. Such artificial keys and key-value pairs reduce the reliability of key-value-based analysis.

Another challenge with the Augeas output stems from the indices assigned to the keys based on the ordering of entries in the file. Due to these indices, a different parameter ordering can result in a different key set. For example, in the httpd configuration file in Fig. 5, if the second and the third lines were swapped, `/directive[2]` and `/directive[3]` keys would have referred to `Redirect` and `Listen`, respectively, unlike the Augeas output in Fig. 5.

Because of the problems described above, Augeas key-value pairs are often *ambiguous*, where the values of keys do not necessarily correspond to the values of configuration parameters. For example, the value of the key `/directive[1]` in Fig. 5 is `Listen`, which is a configuration parameter rather than a configuration value such as `8080`. Moreover, Augeas keys do not correspond to the same configuration consistently across different files. Hence, Augeas key-value pairs are not effective for corpus-based configuration analysis[2].

2) Disambiguation of the Augeas Output: Figure 5 depicts the overall flow in our extraction phase. First, we parse the discovered files using Augeas, which discards any file that does not comply with the target application's configuration file format. We then *disambiguate* Augeas' output to generate reliable key-value pairs where a key consistently corresponds to the same single parameter across different files. For this purpose, we convert the Augeas output into an *intermediate tree* that retains the hierarchical information in the configuration file. We transform this tree using a list of application-specific rules such that the transformed tree faithfully represents all parameters in the configuration files. We manually implement these rules using minimal domain knowledge and only by examining the document structure, parameters found in the configuration files, and their corresponding output produced by Augeas.

[2]Although the Augeas keys do not meet the consistency property, they can be used for analysis [17] if the target configuration parameters are already defined by unique keys (such as in PostgreSQL) or if the target files have the same parameter ordering. However, an identical parameter ordering across different configuration files is not guaranteed in a multi-tenant cloud platform where the files are curated by different users.

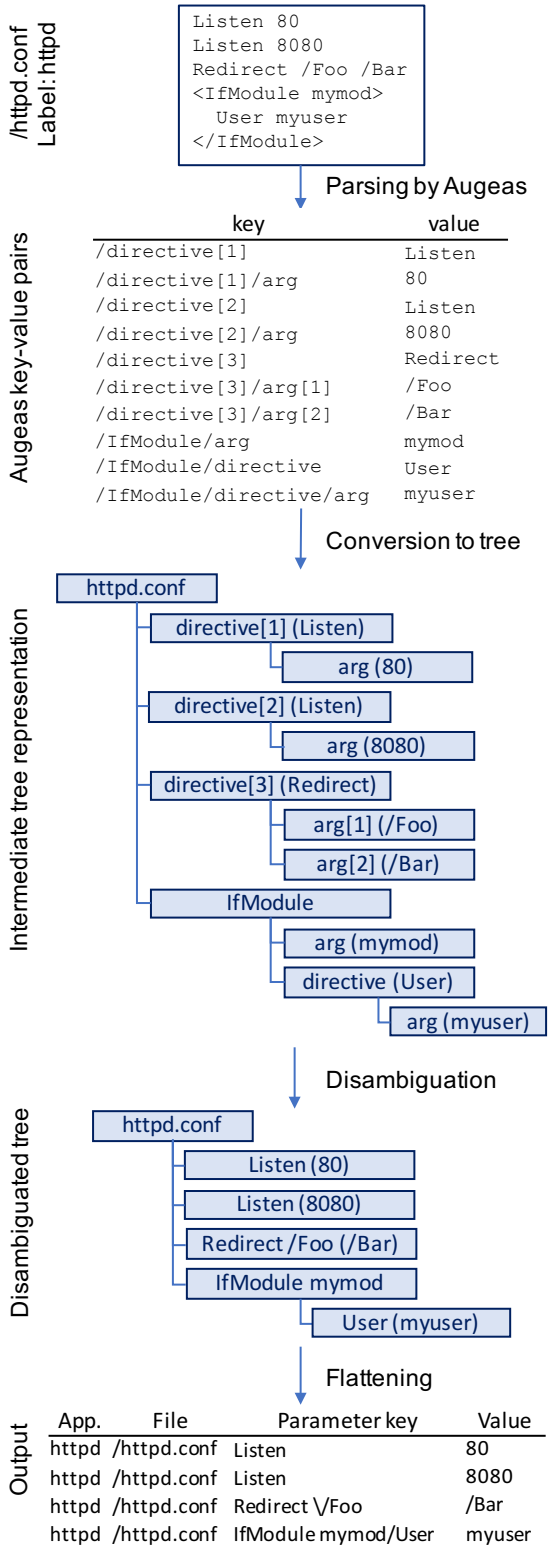


Fig. 5. Extraction phase. Augeas parses configuration files based on the labels given by the discovery phase. The key-value pairs generated by Augeas is converted into a tree that retains the configuration hierarchy. In this tree, texts in parenthesis represents the values of nodes. This tree is *disambiguated* based on application-specific rules that are manually generated using minimal domain knowledge. The flattened form of the disambiguated tree contains key-value pairs where a key corresponds to a single parameter consistently across different files and cloud instances.

Example disambiguation rules: By examining httpd configuration files and the Augeas output, we observe that the `directive` keys are redundant and do not correspond to parameters; hence, we extract the actual parameter names from the values of the `directive` keys. The configuration options assigned to these parameters are extracted from the value of the child node named `arg` in the intermediate tree. We also observe that specific entries such as `Redirect` do not represent parameters, but they are configuration commands with multiple arguments. From a configuration analysis perspective, we are interested in which arguments are being redirected (`/Foo` in Fig. 5) and where they are directed to (`/Bar` in Fig. 5). In this case, we use `Redirect /Foo` as the key, indicating that `/Foo` is being redirected, and `/Bar` as the value assigned to this key. We identify fifteen such configuration commands in httpd by skimming through the application documentation. Our final observation is that nodes without values (such as `IfModule`) indicate configuration hierarchy. We use such nodes to preserve configuration hierarchy without assigning specific values to them. The above observations can be summarized in the following three tree transformation rules for httpd configurations:

- `directive` nodes are replaced by the parameter names stored in the value of the directive. The value of the new node is the value of the child node named `arg`.
- For specific keys that represent configuration commands (such as `Redirect`), the new key is appended with the value of the child node named `arg[1]`. The value of the new node is the concatenation of the values of the remaining children whose name start with `arg`.
- Nodes without values (such as `IfModule`) are converted into an intermediate node where their key is appended with the value of the concatenation of the values of the children whose name start with `arg`.

After this rule-based transformation, the new tree is flattened and converted into a table as depicted in Fig. 5. The application label and the file path are also appended to this table such that all configurations extracted from a cloud instance are represented in a single standardized format for robust analysis.

To extract reliable key-value pairs from the configuration files of new applications, one needs to implement tree transformation rules specific to the new application by examining the configuration file structure, configuration parameters, and the corresponding Augeas output using minimal domain knowledge as described above. A new Augeas lens may be required if the Augeas library does not support the new application.

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 [15], parameter type inference [17], [25], rule-based validation [26], [27], parameter correlation analysis [19], and matching configuration parameters with the parameters in the source code for source-based analysis [28], [29].

TABLE II
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

IV. EXPERIMENTAL METHODOLOGY

We evaluate ConfEx using public images in the Docker Hub repository. To understand the individual benefits of discovery and extraction, we evaluate these two phases separately. In addition, we present two use cases of ConfEx: (1) detecting injected misconfigurations through outlier analysis and (2) syntactic configuration validation.

A. Target Images

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 are downloaded at least 50 times and contain the application name in their name or description. We have manually labeled the application configuration files in these images by examining file contents and file paths. Table II summarizes the number of images we use in our evaluation along with the number of text files and identified configuration files in these images. In total, we use 4581 images, where some images contain both the `/etc/services` file and one of the target applications.

B. Evaluation of Discovery

We compare ConfEx's discovery phase with Augeas' configuration file discovery approach, which is checking the existence of files in specific file paths. Table III shows the paths checked by Augeas to discover httpd configuration files as an example. These file paths account for the default application installation paths in various Linux distributions.

We measure the effectiveness of configuration file discovery separately for each application and using five-fold cross validation. That is, for each application, we randomly divide the images that contain the application into five equal-sized partitions. We use four of these partitions to train our framework by generating an application vocabulary, and all the text files in the images of the fifth partition as testing set, where configuration discovery predicts whether the input text files are configuration files of the target application. We repeat this procedure five times, where each partition is used as a testing set once. Furthermore, we repeat the five-fold cross validation ten times with different randomly-selected partitions.

TABLE III
FILE PATHS CHECKED BY AUGEAS TO IDENTIFY HTTPD CONFIGURATION FILES. "*" IS A WILDCARD THAT REPRESENTS ANY FILE NAME.

/etc/httpd/conf/httpd.conf
/etc/httpd/httpd.conf
/etc/httpd/conf.d/*.conf
/etc/apache2/sites-available/*
/etc/apache2/mods-available/*
/etc/apache2/conf-available/*.conf
/etc/apache2/conf.d/*
/etc/apache2/ports.conf
/etc/apache2/httpd.conf
/etc/apache2/apache2.conf

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.

C. Evaluation of Extraction

The disambiguation process described in Sec. III-B2 significantly impacts the parameter and value distributions observed across the configuration corpus. We demonstrate this impact by studying the total number of distinct values each key gets with and without disambiguation. As the configuration files of the same application may reside in different paths in different images, we analyze all extracted application key-value pairs regardless of the paths of the source files.

D. Detecting Misconfigurations with Outlier Analysis

We present a quantifiable demonstration of ConfEx by automatically detecting injected misconfigurations using PeerPressure [15]. PeerPressure finds the culprit configuration entry in an image with a single configuration error, where configurations are provided as key-value pairs. For each key-value pair, PeerPressure examines the values assigned to the key across a trusted corpus, and calculates the probability of the given value being a misconfiguration based on empirical Bayesian estimation. If a value is an outlier among the values that are assigned to the same key across the corpus, the corresponding entry has a high probability of being misconfigured. Finally, the key-value pairs are ranked based on the calculated probabilities, so that the pairs that are ranked higher are outliers, and hence, the most likely errors.

As PeerPressure is designed for Windows registry, it does not have configuration discovery and extraction capability. We use ConfEx to generate configuration key-value pairs for PeerPressure's outlier analysis. Additionally, we show the impact of ConfEx's key-value pair disambiguation on PeerPressure's accuracy by using the default Augeas key-value pairs before disambiguation as a baseline.

We use PeerPressure to detect the application misconfigurations listed in Table IV as well as synthetic `/etc/services` misconfigurations. For applications, we inject each misconfiguration listed in Table IV 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. For each injection, we train PeerPressure using the key-value pairs that belong to the target application from all images except for the misconfigured image. We then run PeerPressure and record its output ranking for the injected error.

TABLE IV
INJECTED APPLICATION MISCONFIGURATIONS

application	name	description
httpd	url	Error 401 points to a remote URL [30]
httpd	dns	Unnecessary reverse DNS lookups [31]
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 [18]
MySQL	buf	Unusually large sort buffer [32]
MySQL	limit	Too low connection error limit [33]
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 [34]
Nginx	access	Giving access to root directory [35]
Nginx	host	Using hostname in a listen directive [35]

E. Rule-based Configuration Type Validation

We present a second use case of ConfEx by validating configuration types (such as integer or file path) through syntactic rules. For this purpose, we randomly select a configuration file, and for each parameter in this file, we write syntactic type validation rules using regular expressions similar to those used in prior work [17], [25]. For example, if the type of a parameter is IP address, the value assigned to this parameter is validated using the regular expression `^\d{1,3}(\.\d{1,3}){3}$`. For each rule, we map the rule to the key that points to the rule's target value in the selected file. For example, if the selected file is the one presented in Fig. 5, we check whether the value assigned to the keys `/directive[1]/arg` (for Augeas keys) and `Listen` (for ConfEx keys) is a port number across all known configuration files in our corpus.

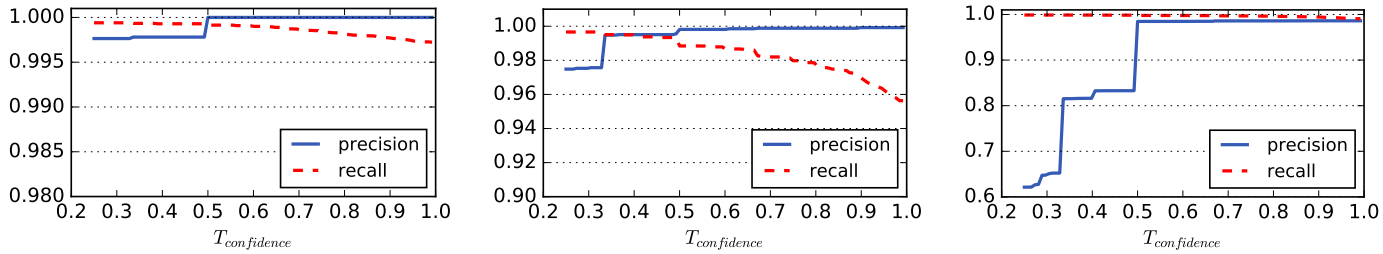
Based on the randomly selected file, we write syntactic rules for 25, 12, and 25 parameters in httpd, MySQL, and Nginx, respectively. We use these rules to validate the values assigned to the corresponding keys, and show the impact of disambiguation on syntactic configuration validation.

V. RESULTS

This section first discusses the selection of the confidence threshold in the discovery phase of ConfEx. Then, we compare ConfEx's discovery and extraction phases with the baseline approaches. Sections V-C and V-D demonstrate use cases of ConfEx to detect configuration errors.

A. Configuration File Discovery

Figure 6 shows the precision and recall ConfEx achieves on identifying the configuration files of the three target applications with various $T_{confidence}$. With $T_{confidence} > 0.5$,



(a) httpd (b) MySQL (c) Nginx

Fig. 6. Configuration file discovery results using vocabulary-based discovery w.r.t. confidence threshold. With a confidence threshold above 0.5, ConfEx's discovery approach achieves above 0.95 precision and recall in all applications we study. Note that the subfigures have different vertical axis scales.

ConfEx achieves above 0.95 precision and recall for all three applications.

The precision of discovery typically increases with the increasing $T_{confidence}$. This is because with a high $T_{confidence}$, the input text files with a few *important* words that don't exist in the vocabulary are labeled as non-configurations, reducing false positives. $T_{confidence}$ has a higher impact on Nginx's precision compared to httpd and MySQL as Nginx uses parameter names and command words that are commonly found in other text files (such as `user` and `include`).

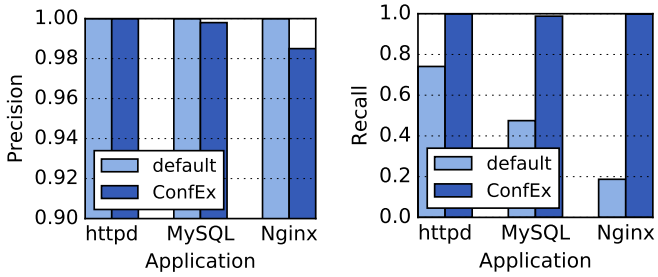
While increasing the precision, the increasing $T_{confidence}$ decreases recall. The impact of $T_{confidence}$ on recall is more obvious in MySQL configuration files, which tend to be diverse where some configuration files contain rarely-used parameters. In addition, MySQL configuration files include fewer configuration commands compared to httpd and Nginx, decreasing the number of words in $\mathbb{W}_{\text{test}}$ (i.e., set of important words in the input text file) that also exist in the vocabulary.

The consistently high precision and recall in Fig. 6a indicates that most of the words in the httpd vocabulary are unique to httpd (such as `LoadModule` and `IfModule`).

In the rest of our study, we use the $T_{confidence} = 0.5$. This prevents misprediction of the non-configuration files that have two words in their $\mathbb{W}_{\text{test}}$ if only one of these words exist in the vocabulary with a $min\_cardinality_w \leq 2$, while considering that an input configuration file's $\mathbb{W}_{\text{test}}$ may contain words that are not seen during the training.

Figure 7 compares the baseline discovery approach of checking the standard configuration file paths with the proposed vocabulary-based discovery. The default approach achieves ideal precision, i.e., all labeled configuration files are correctly labeled without false positives. This is because the standard configuration file paths (such as those shown in Table III) do not contain text files that are not configurations. However, as shown in Fig. 7b, 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



(a) precision (b) recall

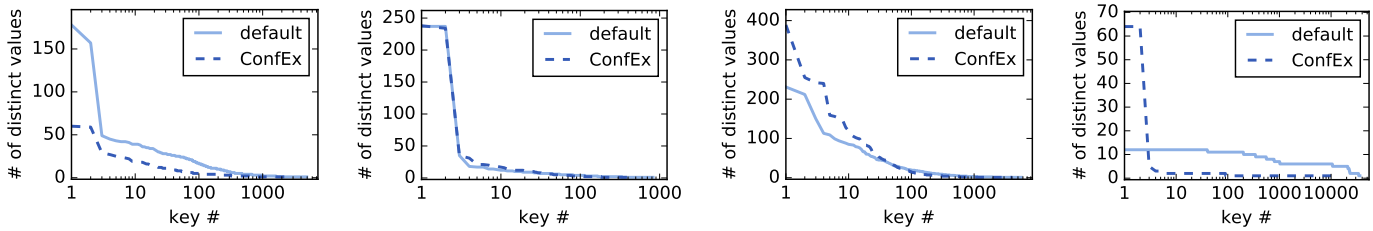
Fig. 7. Comparison of the default path-based and ConfEx's vocabulary-based discovery approaches ($T_{confidence} = 0.5$). The default approach can identify only 19-75% of the application configuration files, leading to low recall. ConfEx successfully identifies more than 98% of these files while resulting in less than 2% false positives in Nginx.

parameter names are uncommon. As these parameter names are not seen during vocabulary generation, the corresponding files are labeled as non-configuration files. ConfEx's lowest precision, which is over 98%, is observed with Nginx, where ConfEx labeled 347 of the non-configuration files as Nginx configurations among over 300,000 unlabeled text files. These mislabeled files contain words that are used as parameter names in Nginx such as the word `include` in file `/etc/ld.so.conf`.

B. The Impact of Disambiguation

In Fig. 8, we focus on the number of distinct values per configuration key across all known configuration files to show how ConfEx's disambiguation step changes the distribution of the extracted key-value pairs.

When disambiguation is applied to the key-value pairs in the httpd configuration corpus, the number of distinct values assigned to individual keys reduce significantly. This is because an Augeas key can correspond to different parameters in different images. For example, `directive[1]` and `directive[2]` can correspond to `Listen` and `Redirect`, respectively, in one file, but `Redirect` and `Listen`, respectively, in another, giving the impression that the `directive[1]` and `directive[2]` keys both have two distinct values. This problem is resolved by ConfEx's



(a) httpd (b) MySQL (c) Nginx (d) `/etc/services`

Fig. 8. The number of distinct values per key across application configuration corpora before and after ConfEx's disambiguation. The keys are sorted individually for each line. Each key corresponds to a single configuration parameter when using the disambiguated keys. The default Augeas keys, however, may correspond to different parameters in different images. Disambiguation substantially changes the distribution of the observed key-value pairs.

disambiguation, where a key consistently corresponds to the same parameter across different files and images.

In Fig. 8b, the impact of disambiguation appears to be less significant for MySQL. However, upon further inspection, we observed that the keys do not correspond to the same parameters. For example, the first two disambiguated keys in Fig. 8b correspond to the MySQL passwords assigned to `client` and `mysql_upgrade` in the configuration file, whereas the first two keys in the default keys correspond to the passwords assigned to `mysql`, `connector_python`, and `mysqladmin` in addition to `client` and `mysql_upgrade`.

The Nginx distributions shows a decrease in the number of distinct values per key after disambiguation. The first key in both default and disambiguated distributions, `server/server_name`, illustrates how the number of distinct values per key decreases. When using the disambiguated keys, the key `server/server_name` covers all Nginx server name entries across the images. However, with the default keys, this key is used only if there is a single `server/server_name` is declared in the configuration file. If there are multiple server names in the same file, Augeas assigns an index to the key (`server/server_name[1]`, `server/server_name[2]`, etc.), preventing the comparison of server names across images using the same key.

Applying disambiguation to network services configurations reveals an interesting fact that is not visible when using the default Augeas key-value pairs: There is a single service, `X11`, that uses more than 60 ports for both tcp and udp connections. All other services use at most two ports.

C. Detecting Misconfigurations with PeerPressure

Given an image with an injected misconfiguration, PeerPressure ranks all the configuration key-value pairs in the image with respect to their probability of being an error. Figure 9 shows the fraction 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 as discussed in Sec. V-B. This problem becomes more significant

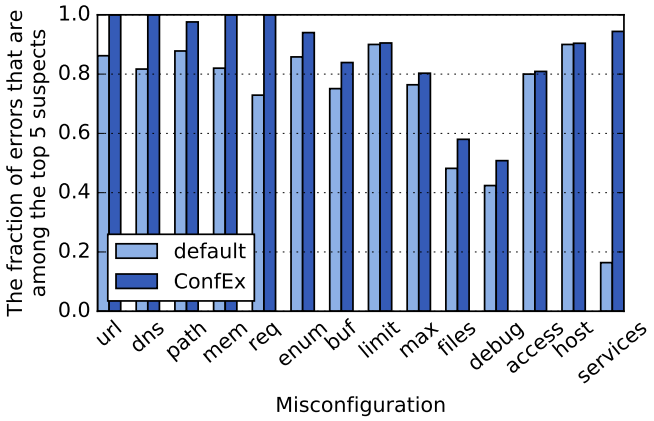


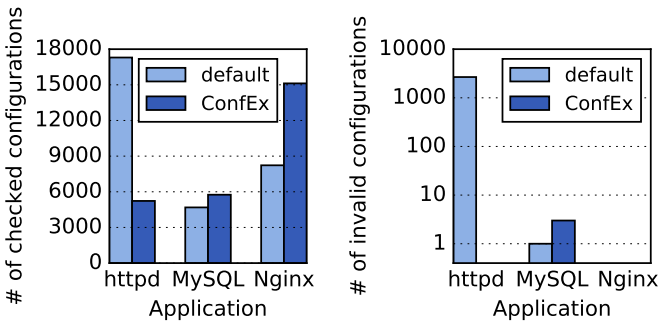
Fig. 9. The fraction of injected errors that are ranked within the top five suspects by PeerPressure among 1000 randomized injections for each misconfiguration. `services` is the `/etc/services` misconfiguration.

when the number of keys are used for 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 than the files seen in the corpus, the parameters in the image are represented by keys that use different indexing. As a result, common configuration entries become outliers in the corpus and have high PeerPressure rankings. However, PeerPressure can still detect the injected errors with the default keys if the parameter 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 a flatter value distribution with a large number of distinct values across the corpus. Hence, the injected erroneous value is not perceived as an outlier by PeerPressure. This is an inherent weakness of outlier analysis and can be avoided by using a larger configuration corpus.

D. Rule-based Type Validation

Fig. 10a shows the number of configurations that use the keys for which we have written syntactic validation rules



(a) Number of checked configurations (b) Number of values marked as invalid

Fig. 10. The number of configuration entries checked and marked as invalid using rule-based type validation. With the default keys, all values that share the same key are checked using the same rule although they belong to different parameters in httpd, resulting in a high number false negatives. In MySQL and Nginx, default Augeas keys can capture only a subset of the target key-value pairs that can be validated by the given rules.

for both default Augeas and ConfEx's disambiguated keys. As the same default Augeas key can correspond to multiple parameters in httpd, all values that share the same key are checked using the same rule even if they correspond to different parameters. This results in over 2500 values being marked as invalid as seen in Fig. 10b. The same problem does not occur with MySQL and Nginx as their Augeas keys are not shared by different parameters. However, when using the default keys, the validation misses 1068 (19%) and 6894 (46%) of the target MySQL and Nginx parameters, respectively.

With ConfEx's keys, the validation rules detect only three invalid values, and one of these values is also captured with the default keys. We have found that these values are to be replaced by a script (e.g., one of the values is `__PORT__`), and are indeed syntactically invalid.

VI. Related Work

Finding and preventing errors is a major focus of the research on software configurations. Execution trace analysis and binary instrumentation have been shown to provide insight on the root causes of configuration errors [12], [36]. Due to their intrusiveness, however, instrumentation and trace analysis are often impractical on production workloads. Some techniques can validate configurations before deployment to avoid service disruptions and outages. Source code analysis [18], [28], [29], [37] and natural language processing on application documentations [14], [38] have been used to infer configuration constraints. Configuration entries that do not comply with these constraints are then marked as errors. Application-agnostic statistical techniques such as PeerPressure [15], EnCore [17], and ConfigV [16] use previously observed configurations to learn about the common patterns, and identify deviations as potential errors. These techniques require key-value pairs that represent configurations for analysis and validation, and do not address discovery or extraction for text-based configuration files.

Recently, Huang et al. proposed SAIC [39], a tool to help users discover text-based configuration files in cloud instances with unlabeled content. To identify configuration files, SAIC analyzes the change patterns of files over the lifetime of a cloud instance. Hence, SAIC is only applicable to cloud instances that have multiple versions where the configuration file locations remain the same and configurations are modified.

In prior studies, the extraction of configuration key-value pairs have performed using several methods: Parsing known configuration files with custom scripts (e.g., [14]), crawling erroneous files from mailing lists and technical forums (e.g., [11]), parsing files located in default paths using configuration parsing libraries (e.g., [17]), and using standardized configuration stores such as Windows registry (e.g., [15]). In image repositories and multi-tenant cloud environments, however, configuration file locations are unknown, and configuration parsers produce key-value pairs that lack the consistency and robustness required for meaningful statistical analysis.

Existing tools for handling configurations focus on centralized management rather than extracting key-value pairs in a cloud environment. Tools such as Chef [40] and Ansible [41] have configuration editing capabilities that are restricted to search-and-replace based on regular expressions, but they cannot extract configuration data from text files. CFEngine [42] can parse standard file formats such as XML and JSON, but not application-specific files formats such as in httpd and Nginx configurations. Several tools, including Puppet [23] and bcfg2 [24], can edit application-specific files by leveraging Augeas library [22]. As we show in this work, using Augeas library alone is not sufficient for parameter extraction for a robust corpus-based analysis.

To the best of our knowledge, our work is the first to introduce a configuration analytics framework to enable discovery and analysis of configurations in image repositories and multi-tenant cloud platforms.

VII. Conclusion

In this work, we have proposed 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 loosely-structured text-based configurations in the cloud. To discover configuration files in cloud instances with unlabeled content, ConfEx keeps track of the words that appear as the first word of non-comment lines in text files. To parse these files, ConfEx leverages a community-driven configuration parser, Augeas. It then disambiguates the key-value pairs generated by Augeas to achieve key-value pairs that represent application configuration parameters and are consistent across different files and cloud instances. Our results have shown that ConfEx achieves over 98% precision and recall on identifying configuration files and consistently improves the efficacy of detecting configuration errors through outlier analysis and syntactic validation.

REFERENCES

[1] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE, 2015, pp. 307–319.

[2] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury, "Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications," in *Proceedings of the 6th International Conference on Autonomic Computing*, ser. ICAC, 2009, pp. 169–178.

[3] R. Johnson. (2010) More details on today's outage. https://goo.gl/3QOVUn.

[4] L. Y. Liang. (2013) Linkedin.com inaccessible on thursday because of server misconfiguration. https://goo.gl/FBa57m.

[5] Y. Sverdlik. (2012) Microsoft: misconfigured network device led to azure outage. https://goo.gl/WhbvzK.

[6] K. Thomas. (2011) Thanks, amazon: The cloud crash reveals your importance. https://goo.gl/CVOsMX.

[7] M. Welsh. (2013) What i wish systems researchers would work on. https://goo.gl/eCzVNj.

[8] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *USENIX symposium on internet technologies and systems*, vol. 67. Seattle, WA, 2003.

[9] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE Software*, vol. 30, no. 4, pp. 88–94, 2013.

[10] 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 23rd ACM Symposium on Operating Systems Principles*, ser. SOSP, 2011, pp. 159–172.

[11] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *12th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2016.

[12] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI, 2012, pp. 307–320.

[13] 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*, ser. ESEC/FSE, 2015, pp. 295–306.

[14] R. Potharaju, J. Chan, L. Hu, C. Nita-Rotaru, M. Wang, L. Zhang, and N. Jain, "Confseer: Leveraging customer support knowledge bases for automated misconfiguration detection," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1828–1839, Aug. 2015.

[15] 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*, ser. OSDI, 2004, pp. 17–17.

[16] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *Proc. ACM Program. Lang.*, vol. 1, pp. 64:1–64:20, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133888

[17] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2014, pp. 687–700.

[18] 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*, ser. SOSP, 2013, pp. 244–259.

[19] W. Chen, H. Wu, J. Wei, H. Zhong, and T. Huang, "Determine configuration entry correlations for web application systems," in *IEEE 40th Annual Computer Software and Applications Conference*, ser. COMPSAC, vol. 1, June 2016, pp. 42–52.

[20] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIXATC, 2011, pp. 28–28.

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

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

[23] J. Loope, *Managing Infrastructure with Puppet: Configuration Management at Scale*. "O'Reilly Media, Inc.", 2011.

[24] N. Desai, "Bcfg2: A pay as you go approach to configuration complexity," *Australian Unix Users Group (AUUG2005)*, vol. 10, 2005.

[25] W. Li, S. Li, X. Liao, X. Xu, S. Zhou, and Z. Jia, "Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'17, 2017, pp. 88–97. [Online]. Available: http://doi.acm.org/10.1145/3084226.3084244

[26] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou, "Confvalley: A systematic configuration validation framework for cloud services," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, 2015, pp. 19:1–19:16.

[27] S. Baset, S. Suneja, N. Bila, O. Tuncer, , and C. Isci, "Usable declarative configuration specification and validation for applications, systems, and cloud," in *Proceedings of the Industrial Track of the 18th International Middleware Conference*, ser. Middleware Industry'17.

[28] S. Zhou, S. Li, X. Liu, X. Xu, S. Zheng, X. Liao, and Y. Xiong, "Easier said than done: Diagnosing misconfiguration via configuration constraints analysis: A study of the variance of configuration constraints in source code," in *International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'17, 2017, pp. 196–201.

[29] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, "Confmapper: automated variable finding for configuration items in source code," in *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2016, pp. 228–235.

[30] T. Osbourn. Cannot use a full url in a 401 errordocument directive - ignoring! http://tosbourn.com/notice-cannot-use-a-full-url-in-a-401-errordocument-directive-ignoring/.

[31] Apache core features. https://httpd.apache.org/docs/2.4/mod/core.html.

[32] Optimize mysql configuration with mysql tuner. https://stackoverflow.com/questions/25166602.

[33] MySQL 5.7 reference manual. https://dev.mysql.com/doc/refman/5.7/en/blocked-host.html.

[34] Debugging nginx. https://www.nginx.com/resources/admin-guide/debug/.

[35] Nginx pitfalls and common mistakes. https://www.nginx.com/resources/wiki/start/topics/tutorials/config_pitfalls/.

[36] S. Zhang and M. D. Ernst, "Which configuration option should i change?" in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE, 2014, pp. 152–163.

[37] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE, 2014, pp. 140–151.

[38] 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*, ser. ASE, 2014, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/2642937.2643009

[39] Z. Huang and D. Lie, "SAIC: identifying configuration files for system configuration management," *CoRR*, vol. abs/1711.03397, 2017. [Online]. Available: http://arxiv.org/abs/1711.03397

[40] M. Taylor and S. Vargo, *Learning Chef: A Guide to Configuration Management and Automation*. "O'Reilly Media, Inc.", 2014.

[41] L. Hochstein, *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media, Inc., 2014.

[42] M. Burgess and R. Ralston, "Distributed resource administration using cfengine," *Software: practice and experience*, vol. 27, no. 9, pp. 1083–1101, 1997.