ConfEx: A Framework for Automating Text-based Software Configuration Analysis in the Cloud

Ozan Tuncer, Anthony Byrne, Nilton Bila, Sastry Duri, Canturk Isci, and Ayse K. Coskun

Abstract—Modern cloud services have complex architectures, often comprising many software components, and depend on hundreds of configurations parameters to function correctly, securely, and with high performance. Due to the prevalence of open-source software, developers can easily deploy services using third-party software without mastering the configurations of that software. As a result, configuration errors (i.e., misconfigurations) are among the leading causes of service disruptions and outages. While existing cloud automation tools ease the process of service deployment and management, support for detecting misconfigurations in the cloud has not been addressed thoroughly, likely due to the lack of frameworks suitable for consistent parsing of unstandardized configuration files. This paper introduces *ConfEx*, a framework that enables *discovery and extraction* of text-based software configurations in the cloud. *ConfEx* uses a novel vocabulary-based technique to identify configuration files in cloud system instances with unlabeled content. To extract the information in these files, *ConfEx* leverages existing configuration parsers and post-processes the extracted data for analysis. We show that *ConfEx* achieves over 99% precision and 100% recall in identifying configuration files on 7805 popular Docker Hub images. Using two applied examples, we demonstrate that *ConfEx* also enables detecting misconfigurations in the cloud via existing tools that are designed for configurations represented as key-value pairs, revealing 184 errors in public Docker Hub images.

Index Terms—Software configuration, cloud, misconfiguration diagnosis.

1 Introduction

Cloud applications are designed in a highly configurable way to ensure high levels of reusability and portability. To function correctly, securely, and with high performance, these applications often depend on precise tuning of hundreds of configuration parameters [1]. The number of parameters to tune can reach thousands in typical cloud services that consist of multi-tiered software stacks [2].

While configurations are traditionally validated by applications during startup, recent work has shown that, across various software applications in today's cloud, 14-93% of configuration parameters do not have any special code for checking their correctness [3]. 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 leverage third-party software and public cloud images without necessarily having the expertise needed to precisely configure all components of their service. This, combined with the often-immature documentation that accompanies newly-introduced software, makes human error the leading cause of configuration-related failures [4]. In a similar vein, configuration errors have taken their place among the leading causes of cloud software failures [5], [6], [7], and have been reported as causes of service disruptions at Microsoft Azure [8], Amazon EC2 [9], and Google [10].

- *O. Tuncer, A. Byrne, and A. K. Coskun are with the Department of Electrical and Computer Engineering, Boston University, Boston, MA, 02215.*
  E-mail: {otuncer,abyrne19,acoskun}@bu.edu
- *N. Bila, S. Duri and C. Isci are with IBM Research, T. J. Watson Research Center, Yorktown Heights, NY, 10598.*
  Email: {nilton,sastry,canturk}@us.ibm.com

Existing failure avoidance and mitigation mechanisms in the cloud (e.g., redundancy or recovery) are insufficient to handle configuration errors, as configurations tend to affect the entire cloud service rather than a single component such as a VM or a process [5], [11]. Widely-used cloud deployment tools such as Chef [12] and Ansible [13] provide centralized management support for configurations. These tools manage the configuration parameters that are related to deployment and scaling but do not typically validate the parameters that determine functionality and performance. Hence, there is a growing need for support to help analyze and validate software configurations in cloud platforms.

There are several challenges in applying automated configuration analysis in the cloud. First, users often do not store their configurations in standard file system locations in cloud instances (i.e., images, VMs, and containers). Especially in multi-tenant cloud platforms, where there is no platform-wide configuration management mechanism, one needs to *discover* the locations of configurations in a cloud instance to perform analysis. Second, configuration parameters of cloud software are often embedded in human-readable text files, where each software has its own custom file syntax. For automated analysis, the information extracted from these files needs to be represented in a consistent format that allows validation and comparison of individual parameters. Third, cloud instances typically contain multiple configuration files that are tuned for different use cases or software versions. As some of these files are not actively used by the running applications, one needs to determine which configurations are *active* in a cloud instance to avoid false positives while detecting configuration errors. Finally, as many configuration parameters are related to the execution environment [14], environmental information such as file access permissions should be collected from

cloud instances. To address the above challenges, one needs a framework to *discover and extract* consistent configuration information from cloud instances with unlabeled content.

In this paper, we introduce *ConfEx*, a novel software configuration analytics framework that enables robust analysis of text-based software configurations in the cloud. *ConfEx* collects environmental information from the cloud instances, discovers configuration files of known applications in these systems, and parses the discovered files to produce consistent configuration data. By itself, *ConfEx* does not perform any validation on the configuration data it extracts. Instead, it enables the use of existing validation tools originally designed for key-value-based configurations (such as *PeerPressure* [15] and *Encore* [14]) on extracted configuration data. As we demonstrate in our evaluation, without *ConfEx*, these tools have limited applicability in the cloud. Our specific contributions are as follows:

- We propose *ConfEx*, a configuration analytics framework that enables the analysis of text-based software configurations in multi-tenant cloud platforms and image repositories. We provide two examples of *ConfEx* being applied to existing configuration analysis tools to detect misconfigurations.
- As part of *ConfEx*, we develop a method to discover the configuration files in cloud instances with unlabeled content. By identifying configuration keywords (such as parameter names) in an application-agnostic manner, our method achieves over 99% precision and 100% recall on identifying configuration files in 7805 Docker Hub images.
- To enable focusing on configuration files that are actively used by running applications, we develop two methods: Our first method targets VMs and utilizes Linux file timestamps; our second method tracks system calls during application initialization and targets containers.
- We demonstrate that the outputs of existing configuration parsers often lack the consistency and robustness for configuration analysis. To resolve this issue, we design a *disambiguation* methodology, enabling comparison and analysis of configurations among thousands of cloud instances.

2 Background on Cloud Software Configurations

Most cloud applications and system services store their configurations in human-readable text files or in configuration stores such as `etcd` or 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). The remainder of this section explains cloud software configurations in detail and discusses how to use analytics to detect configuration errors.

2.1 Text-based Configurations

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 statement. `User` and `Group` parameters are defined

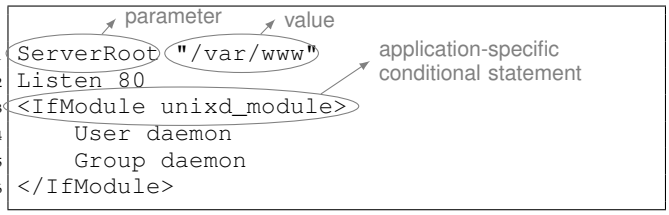


Fig. 1. Httpd configuration file snippet. Configurations are stored in an XML-like format.

```
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.

within the `IfModule unixd_module` section, representing a configuration hierarchy.

In some configuration files, understanding the file schema requires domain knowledge. One such example is the Linux filesystem configuration file (`/etc/fstab`). 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) or multiple comma-separated entries such as `defaults,gid=5` (line 3).

Extracting configuration data from text-based files while retaining the relational information between different entries requires expertise on the specific file format. Hence, to conduct corpus-based configuration analysis on a large number of applications, one should use a parsing tool that is continuously maintained by application domain experts.

2.2 Configuration File Locations

Software package managers such as `rpm` place configuration files to specific file system paths by default. For example, the configurations of MySQL are installed by default to `/etc/my.cnf`, `/etc/mysql/my.cnf`, and `/etc/mysql/conf.d/`.

In cloud platforms, however, users often store their configurations in non-standard locations. By examining popular Docker Hub images as an example, we have discovered that depending on the application, 20-81% of configuration files are located in non-standard paths (see Sec. 4.2 for details). For example, we have identified MySQL configuration files located in `/app/my.cnf`, `/my-mini.cnf`, `/healthcheck.cnf`, and `/usr/cnf`, where file names are not necessarily indicative of MySQL. Hence, one needs a systematic methodology to identify configuration files for comprehensive configuration analysis in the cloud.

2.3 Active Configuration Files

Cloud instances often contain multiple syntactically-valid configuration files for a given application. These files include (1) configurations of modules and plug-ins, which enable new software functionalities, (2) configurations of specific application instances (e.g., Nginx can use a separate configuration file for each *virtual host*), (3) template files, which set the default application behavior and can be directly used, and (4) multiple versions of configuration files that are used for testing and development purposes.

TABLE 1
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.

At the time of deployment, an application will use only a specific set of configuration files, which we refer to as *active* configuration files. These files are determined based on command line options or the main configuration file. The remaining *passive* files typically do not affect the application behavior. Detecting misconfigurations in these passive files would be of little use to cloud users and may be even seen as false positives. Hence, it is necessary to identify which files are actively used in a running cloud instance for accurate error detection.

2.4 Configuration Errors

Table 1 summarizes common misconfiguration types we derived from related work (e.g., [2], [7], [14], [16], [17]) and online technical forums (e.g., stackoverflow.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 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.

2.5 Configuration Analysis

Researchers have developed various tools to automatically check for errors in software configurations (e.g., [18], [19]). Unlike the configurations found in cloud instances, these tools are mainly geared towards configurations that are represented as key-value pairs, where each key consistently corresponds to a specific configuration parameter.

Among automated configuration validation tools, statistical and learning-based techniques (e.g., [15], [20], [21]) have gained popularity as low overhead configuration checkers that can be applied in an application-agnostic manner. These techniques use a corpus of configurations collected from working systems to infer configuration constraints or learn common patterns. Then, configurations that violate the inferred constraints or deviate from the norm are identified as potential errors. Such methods are appealing in practice because they do not require intrusive dynamic analysis or application instrumentation.

Cloud environments, where a large number of users deploy their customized applications, provide a unique opportunity for statistical and learning-based configuration analysis. However, training models for such techniques and using these models to validate configurations requires *discovery* of configurations and *extraction* of configuration parameters across large populations of installed applications.

2.6 Potential Uses for a Cloud Configuration Analytics Framework

As mentioned in the introduction, the cloud's meteoric rise has enabled new highs in developer productivity. Automation, in terms of both cloud operators automating their infrastructure and cloud users automating their application deployments, has been a key driver of this rise [22], [23]. However, a consistent method for automating cloud configuration analysis and validation has not yet been realized, and unless the right application-specific configuration validation tool happens to exist, service reliability engineers are often forced to manually review every revision of each application configuration that they manage.

A cloud-focused configuration analysis framework could lay the groundwork for a solution to this problem. With the permission of cloud users, cloud operators could collect user configurations in VMs and containers to create a large and highly-diverse configuration dataset and to train accurate configuration models. These models could then be used to identify problematic configurations before deployment, saving cloud users days of engineering time that would otherwise be spent on configuration debugging.

3 Configuration Analytics with ConfEx

Our goal is to systematically analyze text-based software configurations in image repositories and multi-tenant cloud platforms, where cloud instances are calibrated by different users and include unlabeled content. To this end, we design a configuration analytics framework, *ConfEx*.

Figure 3 shows the overview of our *ConfEx* framework. *ConfEx* has three phases: *discovery*, *extraction*, and *analysis*. In the discovery phase, we identify the actively used configuration files and collect environmental data from target systems. The extraction phase then parses the information in the configuration files, and transforms the collected data into key-value pairs where each key corresponds to a single configuration parameter. Finally, in the analysis phase, we use existing tools based on outlier detection and rule-based validation to analyze the extracted configuration data. The rest of this section explains these three phases in detail.

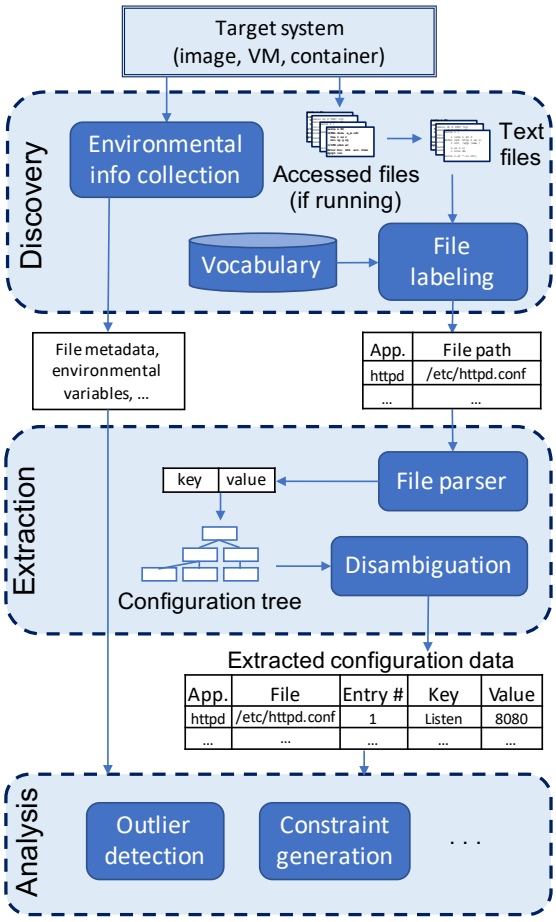


Fig. 3. *ConfEx* overview. The discovery phase collects environmental data and identifies actively used configuration files. The extraction phase parses the information in the identified files and generates configuration data that is consistent across cloud instances. The analysis phase applies existing rule-based, statistical, or learning-based tools to analyze and validate the configurations.

3.1 Discovery

Discovery focuses on (1) identifying configuration files that are actively used in the target system and (2) collecting environmental data (such as user privileges and file permissions) that can be used to detect misconfigurations.

3.1.1 Active File Discovery

As discussed in Sec. 2.3, only a specific set of configuration files are used in a running cloud instance. Including the remaining unused files in configuration analysis can lead to unreliable results, and misconfigurations detected in these unused files may be seen as false positives by cloud users. Hence, *ConfEx* focuses only on the files that are accessed by applications in VMs and containers.

We have developed two solutions to determine the files that are accessed: (1) checking file timestamps, and (2) tracking system calls during application initialization. Our solutions, as described below, differ in their degree of applicability and intrusiveness.

Checking file timestamps: One way to understand whether a file has been accessed is to check its Unix access timestamp, `atime`. In most file systems, `atime` is updated by default when the file is read for the first time after being modified or if the existing `atime` is older than one day. Hence, during discovery, we check `atime`'s and ignore the files that have not been accessed since a specific time point, which is the last system restart time by default and can be overriden by the user. We use `atime` for active file discovery in VMs; however, this approach is not applicable if `atime` is disabled through file system mount options or in copy-on-write file systems such as shared Docker layers or *btrfs* [24].

Tracking system calls: In systems where the `atime`-based approach is inapplicable, we identify accessed files by tracking `open()` system calls that have the `read` flag using the `auditd` tool on the host machine. Note that applications typically read their configurations during initialization; hence, we need to track system calls only during application initialization. This approach is suitable for containers, which typically start with an entry script that runs the application inside the container. In situations where the application being analyzed is known to load configuration files long after its initialization (e.g., a modular web server), the user may configure ConfEx to continue monitoring syscalls for a longer duration.

Both active file discovery solutions are only applicable to cases where the target systems are actively running. In cases where the target system or image must be analyzed offline, active discovery can be skipped, and instead all available files can be passed to the identification stage.

3.1.2 Configuration File Identification

The files identified by the active file discovery step include binaries and data files along with configuration files. To identify the configuration files among the accessed files, we first discard non-text files. Second, to reduce computational overhead, we discard the files with extensions that are used for non-configuration files (such as `.h` or `.md5sums`) and the files that are larger than an empirically-determined size threshold (see Sec. 4 for details). To identify configuration files among the remaining text files, *ConfEx* examines the content of these files in the *file labeling* step in Fig. 3.

Figure 4 depicts *ConfEx*'s file labeling step in detail. During offline training, we use known configuration files that are labeled with application names. We then identify the *configuration keywords* in these files to generate application-specific vocabularies. Configuration keywords include parameter names and configuration commands, and are usually specific to applications. We extract these keywords in an application-agnostic way as follows: We first discard commented-out lines, i.e., lines that begin with `//`, `#`, or `%`, excluding the preceding white-space characters. 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. Hence, we use the first word of the remaining lines as keywords. While extracting keywords, we use the following characters as delimiters to account for the characters that are commonly used as part of a configuration file syntax: `\t`, `=`, ` `, `:`, `<`, `>`, `[`, `]`, and `,`. An application vocabulary consists of sets of unique keywords for each known configuration file.

During testing, we again extract the keyword set in the input text file using the same methodology. We calculate the similarity of the input keyword set to each keyword set in the vocabulary of each application. To calculate the

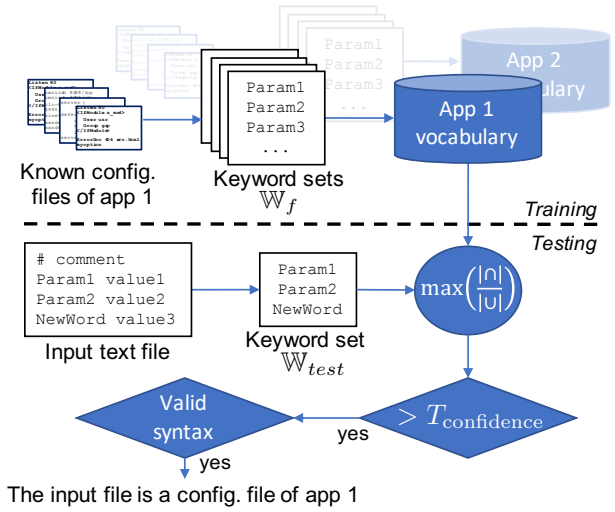


Fig. 4. File labeling step of the discovery phase. During offline training, a vocabulary is generated for each application using known configuration files. Input text files are compared with each application vocabulary. Upon a match that is larger than a confidence threshold, syntactically valid files are labeled as configuration files.

similarity of a set pair, we use the Jaccard index [25], defined as $J = |\mathbb{W}_1 \cap \mathbb{W}_2|/|\mathbb{W}_1 \cup \mathbb{W}_2|$, where $\mathbb{W}_1$ and $\mathbb{W}_2$ are two sets. If the maximum achieved similarity using the keyword sets in an application vocabulary is lower than an empirically-determined threshold, $T_{confidence}$, the file is discarded (see Sec. 4.2 for details). Finally, we check the syntax of the files with sufficient keyword set similarity, and label syntactically valid files as application configuration file.

Note that calculating the Jaccard index between the input keyword set, $\mathbb{W}_{test}$, and keyword sets for all known configuration files is computationally expensive. Furthermore, most non-configuration files do not contain any application-specific keywords and need not to be compared with all keyword sets in a vocabulary. Hence, we speed-up set comparison as follows: Let a keyword set of a known configuration file f be $\mathbb{W}_f$, and the union of all keywords in a vocabulary be $\mathbb{W}_{vocab}$. Then, the Jaccard similarity (J) has the following upper bound:

$$J = \frac{|\mathbb{W}_{test} \cap \mathbb{W}_f|}{|\mathbb{W}_{test} \cup \mathbb{W}_f|} \leq J_{upper} = \frac{|\mathbb{W}_{test} \cap \mathbb{W}_{vocab}|}{|\mathbb{W}_{test}|} \quad (1)$$

as $\mathbb{W}_f \subseteq \mathbb{W}_{vocab}$, and $|\mathbb{W}_{test} \cup \mathbb{W}_f| \geq |\mathbb{W}_{test}|$. Checking J_{upper} once per vocabulary eliminates the need to compare $\mathbb{W}_{test}$ with all $\mathbb{W}_f$'s in the vocabulary if $J_{upper} < T_{confidence}$.

To add a new application or extend an application's existing vocabulary, one can simply process new configuration files that are labeled with application names without the need of re-processing all known configuration files.

3.1.3 Collecting Environmental Data

As mentioned in Sec. 2.4, misconfigurations can occur due to a mismatch between software configurations and environmental settings such as IP addresses and file permissions. Table 2 lists the environmental information *ConfEx* collects. All the information we collect can be used for analysis on VMs and containers; all except the network address and the active port information can be used for analysis on images.

TABLE 2
Environmental information collected from cloud instances.

Description	Source
User information	`/etc/passwd`
Group information	`/etc/group`
File metadata	Crawling the file system
Environmental variables	`docker inspect` or `env`
Network addresses	`docker inspect` or `ifconfig`
Active ports	`docker inspect` or `netstat`

3.2 Extraction

The extraction phase parses the configuration data located in text files identified by the discovery phase, and generates key-value pairs that represent configurations. Such key-value pairs can be directly used by the existing configuration analysis tools such as Encore [14] and ConfigV [21].

While existing studies on configuration analysis have mostly focused on configuration stores that do not require data extraction such as Windows Registry (e.g., [26]), or configurations with standard file formats such as `XML` or `JSON` (e.g., [19], [27]), most configuration files of cloud applications are kept in human-readable text files that do not use standard file formats. As discused in Sec. 2.1, 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 and maintain custom parsers for different applications for every configuration analysis tool.

3.2.1 Augeas for Parsing Configuration Files

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 [28], which is one of the most popular tools 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, and PostgreSQL. Augeas has been maintained for more than ten years, has interfaces in different programming languages including Python, Ruby, and Java, and used by other configuration management tools including Puppet [29] and bcfg2 [30].

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 the cloud. The issues with the Augeas parser output can be seen in the example in Fig. 5. Augeas produces artificial keys (e.g., `/directive[1]`) that do not correspond to parameters but 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. For example, in the httpd configuration file in Fig. 5, if the first two lines were swapped, `/directive[1]` and `/directive[2]` keys would have referred to `Redirect` and `Listen`, respectively, unlike the Augeas output in Fig. 5. Because of this *ambiguity* of Augeas key-value pairs, directly using Augeas is often ineffective for corpus-based configuration analysis.

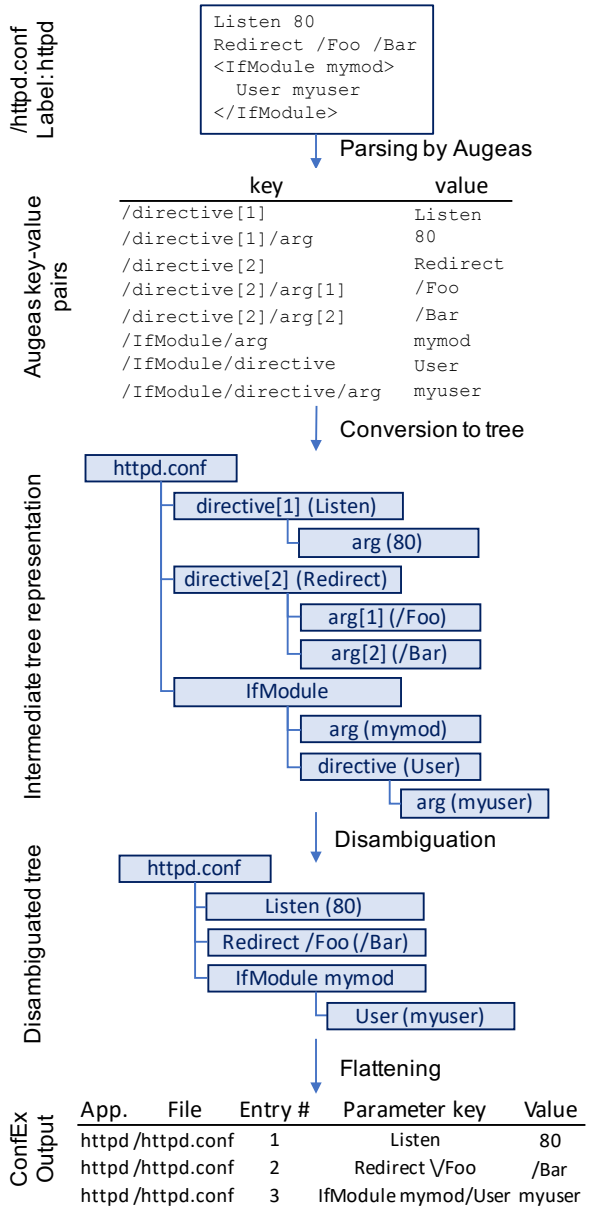


Fig. 5. Extraction phase. Augeas parses configuration files based on the labels given by the discovery phase. The key-value pairs generated by Augeas is converted into a tree that retains the configuration hierarchy, where texts in parenthesis represent the values of nodes. This tree is *disambiguated* 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.

3.2.2 Disambiguation of the Augeas Output

To prepare Augeas' output for corpus-based analysis, we use a *disambiguation* step and transform Augeas' output into key-value pairs where a key consistently corresponds to the same single parameter across different files. As depicted in Fig. 5, we convert Augeas's output into an *intermediate tree* that retains configuration hierarchy. We transform this tree using a list of application-specific rules such that the transformed tree faithfully represents all configuration parameters. 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.

Example disambiguation rules: By examining httpd configuration files and the Augeas output, we observe that `directive` keys are redundant, and 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`. We also observe that specific entries such as `Redirect` represent 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 15 such configuration commands in httpd documentations. Our final observation is that nodes without values (such as `IfModule`) indicate configuration hierarchy. These observations can be summarized in the following transformation rules for httpd:

- `directive` nodes are replaced by the parameter names stored in the node's value. 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 disambiguated tree is flattened and converted into a table as depicted in Fig. 5. In this table, the entry number represents the ordering of the values in the configuration file. 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 analysis.

To extract reliable key-value pairs from the configuration files of a new application, 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.

3.3 Analysis

The discovery and extraction phases of *ConfEx* produce consistent key-value pairs that represent software configurations along with environmental information from the cloud instances, enabling the use of a rich variety of configuration analysis and validation techniques in multi-tenant cloud platforms and image repositories. Analysis of software configurations can be used both to detect misconfigurations and to gain insight on user configuration practices. Existing automated misconfiguration detection techniques that can be applied as part of *ConfEx* include outlier value detection [15], parameter type inference [14], [31], rule-based validation [32], [33], parameter correlation analysis [17], matching configuration parameters with the parameters in

TABLE 3
Statistics on the studied Docker Hub Images

target application	# of images that contain the app.	total # of app. config. files
httpd	1601	53100
MySQL	2238	9481
Nginx	3714	32343
Network services	6106	6106
Users & Groups	7805	7805

the source code for source-based analysis [34]. We provide examples of how two of these techniques could be used during this phase in Sec. 5.

3.4 Implementation

We have implemented our *ConfEx* framework using Python. *ConfEx* crawls cloud instances using IBM's public agentless system crawler[1] and uses Augeas 1.7 for file parsing. We have implemented disambiguation rules for httpd, MySQL, and Nginx applications as well as `/etc/services`, `/etc/passwd`, and `/etc/group` system configurations.

4 Evaluation

We evaluate *ConfEx* using the Docker Hub repository, which is one of the largest publicly available container image repositories with over 650,000 registered users [35]. We focus on the Docker Hub images that are either among the most downloaded 2000 images or contain one of the three following popular cloud applications: httpd, MySQL, and Nginx. 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 configuration files in these images by examining file contents and paths of all text files that comply with the application configuration file syntax. In addition to application configuration files, we use the following system configuration files in our evaluation: the network services file (`/etc/services`), the users file (`/etc/passwd`), and the groups file (`/etc/group`).

Table 3 summarizes the number of images we use along with the number of identified configuration files in these images. In total, we use 7805 images, where 1163 images contain configuration files of more than one target application and 254 images contain configuration file of all three applications. The largest three configuration files have the sizes 140KB, 99KB, and 41KB. Hence, we set the file discovery size threshold in the configuration file identification step (Sec. 3.1.2) to 200KB. Our data set contains a total of over 22 million text files that are smaller than 200KB.

The remainder of this section first discusses our findings based on our study of the active files in Docker containers. We then study the impact of $T_{confidence}$ on configuration file discovery, compare *ConfEx*'s discovery phase with the baseline approaches, and discuss the overhead of *ConfEx*.

4.1 Active File Discovery

To identify accessed (i.e., active) files in Docker containers, *ConfEx* tracks `open()` system calls to the container file system during application initialization. In our experiments, we find that tracking `open()` calls for ten seconds is sufficient to capture configuration file accesses of the applications that are already installed in the images. After ten seconds, we observe either no more `open()` calls or periodic calls to specific non-configuration files.

1. https://github.com/cloudviz/agentless-system-crawler

TABLE 4
Statistics on the active configuration files in the selected Docker Hub images

application	# of images	# of accessed files	# of accessed config. files
httpd	50	2415 (out of 559K)	496 (out of 1768)
MySQL	50	10152 (out of 1.1M)	143 (out of 290)
Nginx	50	5788 (out of 404K)	166 (out of 650)

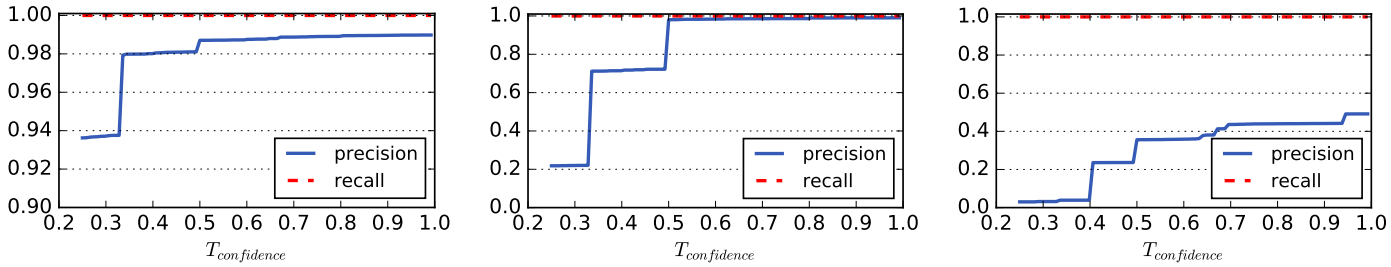
The arguments used while deploying a Docker container may affect which configuration files are used by the running applications. However, most images on Docker Hub do not contain instructions on their intended deployment procedure. Hence, we study the *active file discovery* phase of *ConfEx* using a subset of our target images where we use the necessary commands to correctly initialize the applications.

Table 4 shows statistics on the active files in the selected Docker Hub images. On average, less than 1% of the existing files are accessed during application initialization. This indicates that most Docker images are loaded with extra files that are not needed by the applications. We find that while using these publicly available images significantly reduces development time, these images often contain files and features that are not needed by the final service, filling the deployed container with unnecessary files. For example, the `ubuntu:xenial` image, which a naïve developer may use as a base image, contains over 1000 user manual files, which are not used in automatically deployed containers, and over 500 files for the `perl` package, which may not be needed by the running applications.

Our results in Table 4 also show that only 30% of the syntactically valid configuration files are used by the running applications. As discussed in Sec. 2.1, the remaining configuration files consist of configurations of unused modules and plug-ins, template files, and configuration files that are used for testing and development purposes rather than for deployment. Errors detected in these unused files can be perceived as false positives by cloud users. *ConfEx* prevents such false positives by focusing on active files in VMs and containers. However, for static images, no file access information is available. Thus, configuration analysis should be performed on all configuration files in images.

4.2 Configuration File Identification

We measure the effectiveness of configuration file identification separately for each application and using five-fold cross validation. That is, for each application, we randomly divide the images in our corpus into five equal-sized partitions. We use the configuration files of the target application in four of these partitions to train our framework, and all the text files of the fifth partition as testing set, where we predict 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 five times with different randomly-selected partitions.



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

Fig. 6. Configuration file identification results w.r.t. confidence threshold using only the vocabulary based identification (without syntax check). Recall remains nearly ideal for all confidence thresholds as configuration files use the same set of commands and parameter names. With a confidence threshold above 0.5, *ConfEx* achieves above 0.98 precision for httpd and MySQL even without syntax check.

We use *precision* and *recall* as evaluation metrics. Precision is the fraction of true positives (i.e., correctly identified 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.

4.2.1 Selecting the Confidence Threshold

Figure 6 shows the precision and recall *ConfEx* achieves on identifying the configuration files with various $T_{confidence}$ levels only using the vocabulary-based identification (i.e., without syntax check). Recall remains ideal for all three applications and all $T_{confidence}$ levels. This is because the set of configuration keywords for a given application is limited; and hence, the configuration files in the training and testing sets use the same set keywords. Precision typically increases with the increasing $T_{confidence}$. With a low $T_{confidence}$, the input text files with keywords that don't exist in an application vocabulary are labeled as configurations, increasing false positives. As we show in the next section, nearly all such false positives are eliminated by checking the syntax of the selected files. $T_{confidence}$ has a higher impact on Nginx's precision compared to httpd and MySQL as Nginx uses configuration keywords that are commonly found in the configuration files of other applications and system software (such as `user` and `include`). To account for configuration keywords that may not be observed during training, we set $T_{confidence}$ to 0.9. However, in our dataset, we observe no difference in the file identification results after syntax check for $T_{confidence}$ between 0.8 and 1.0.

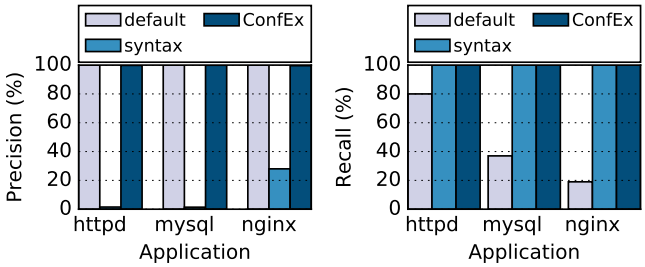
4.2.2 Comparison with Baselines

We implement two baselines using the Augeas configuration editing library [28] to compare *ConfEx*'s configuration file identification accuracy with. Our first baseline, *default*, uses Augeas' discovery approach of checking the existence of files in specific file paths. These paths account for the default application installation paths in various Linux distributions. Table 5 shows the paths checked by Augeas to identify httpd configuration files as an example. Our second baseline, *syntax*, attempts to parse all text files using Augeas, and marks the files that conform with the target application's configuration file syntax as configuration files.

Figure 7 compares the precision and recall of the baselines and *ConfEx* on identifying application configuration files. As all files found in default configuration paths are

TABLE 5
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`



(a) Precision (b) Recall

Fig. 7. Configuration file identification results. While the default approach of checking standard file paths misses up to 81% of configuration files, only checking file syntax leads to low precision by labeling non-configuration files as configurations. *ConfEx*'s file labeling achieves over 99% precision and 100% recall.

indeed application configurations, *default* achieves ideal precision. However, *default* misses 20-81% of configuration files, leading to low recall. *syntax* achieves ideal recall as all application configuration files conform with the expected file syntax. However, a significant portion of non-configuration files are also syntactically valid, leading to low precision. For httpd and MySQL, where the configuration syntax accepts key-value pairs with a space delimiter, the precision of *syntax* is below 2%. For Nginx, where the configuration syntax involves the use of semicolons and curly brackets, the precision still remains below 29%.

By combining the vocabulary-based check and syntactic validation, *ConfEx* achieves over 99% precision and ideal recall for all three applications. The majority of the remaining mislabeled files are configuration files of different applications and have a single configuration command with a keyword `Include` or `include`. These files also have the

correct syntax and would be accepted as valid configurations by our target applications.

4.3 Computational Overhead

Crawling the contents of an image with *ConfEx* takes eight seconds on the average using an Intel Xeon E5-2650 processor. During crawling, the contents of all text files that are smaller than 200KB are compressed and recorded, excluding the files with extensions that are commonly used for non-configuration files (such as `.h` or `.css`). Note that the contents of an image is crawled before deployment, avoiding any performance overhead on applications. Identifying configuration files in an image and extracting the information in these files take 3.5 seconds on the average, where the mean number of text files per image is above 2800.

The only performance impact on a production system is incurred during active file discovery. *ConfEx* uses the `auditd` tool to track `open()` system calls in containers during application initialization, which is less than ten seconds in our experiments. We observe that typically, at most a few hundred `open()` calls are issued during this initialization period, resulting in negligible performance overhead.

5 Applied Examples

ConfEx enables the use of existing key-value-based configuration analysis tools in the cloud. To demonstrate this, we present applied examples of *ConfEx* for detecting misconfigurations using two techniques proposed in prior work: *PeerPressure* [15] and *Encore* [14].

5.1 PeerPressure

PeerPressure [15] is a tool that finds the culprit configuration entry in a Windows image with a single configuration error. *PeerPressure* is designed for Windows registry, where configurations are represented as key-value pairs; and hence, it is not directly applicable on text-based software configurations found in cloud instances. In this example, we show how *ConfEx* and *PeerPressure* can be used together to detect misconfigurations in Docker Hub images, and study the impact of *ConfEx*'s file labeling (Sec. 3.1.2) and disambiguation (Sec. 3.2.2) steps on the effectiveness of *PeerPressure*.

PeerPressure has an offline training and an online testing phase. During training, *PeerPressure* records the histogram of values assigned to each key in a trusted configuration corpus. Given a new image during testing, *PeerPressure* compares the values assigned to each key in the image with the value histograms seen during training. For each key-value pair, it then calculates the probability of being a misconfiguration based on empirical Bayesian estimation. Here, an outlier value has a high probability of being an error. Finally, the key-value pairs are ranked based on the calculated probabilities, so that the pairs that are ranked the highest are the most likely misconfigurations.

We use *PeerPressure* to detect the application misconfigurations listed in Table 6. We inject each misconfiguration to a randomly selected image that contains the target parameter to be misconfigured, and repeat the randomized injection 1000 times. For each injection, we train *PeerPressure* using the configuration key-value pairs extracted from all images except for the misconfigured image. We then run *PeerPressure* and record its output ranking for the injected error.

TABLE 6
Injected application misconfigurations

application	name	description
httpd	url	Error 401 points to a remote URL [36]
httpd	dns	Unnecessary reverse DNS lookups [37]
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 [16]
MySQL	buf	Unusually large sort buffer [38]
MySQL	limit	Too low connection error limit [39]
MySQL	max	Invalid value for max # of connections
Nginx	files	Too few open files are allowed per worker
Nginx	debug	Logging debug outputs to a file [40]
Nginx	access	Giving access to root directory [41]
Nginx	host	Using hostname in a listen directive [41]

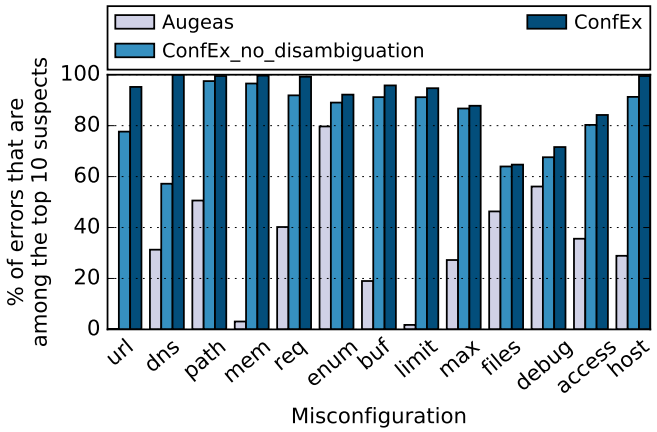


Fig. 8. The percentage of injected errors that are ranked within the top ten suspects by *PeerPressure* among 1000 randomized injections for each misconfiguration. Using *ConfEx*'s disambiguated key-value pairs consistently improves *PeerPressure*'s effectiveness on error detection.

We use two baselines for comparison, where we train and test *PeerPressure* using the configuration key-value pairs provided by the baselines. The first baseline, *Augeas*, parses the configuration files located in default paths using Augeas library, and directly uses Augeas library's output. The second baseline, *ConfEx_no_disambiguation*, again uses Augeas library's output but parses all application configuration files in a given image. We use this baseline to focus on the impact of disambiguation.

5.1.1 PeerPressure Error Detection Results

Figure 8 shows the percentage of injected errors that are ranked among the top 10 suspects by *PeerPressure*. Using *ConfEx*'s disambiguated output consistently leads to similar or higher rankings compared to using the key-value pairs generated by baselines, making it easier to pinpoint the error. 9 out of 13 errors are ranked within the top 10 suspects for more than 90% of the injections.

As the *Augeas* baseline only parses the files located in standard paths, *PeerPressure* cannot find an injected error with this baseline if the error is injected in a file in a non-standard location. The parameters that are modified in the *url*, *mem*, and *limit* misconfigurations rarely appear in files located in standard paths in our corpus; hence, *PeerPressure* misses these errors with the *Augeas* baseline.

When using the key-value pairs generated by the *ConfEx_no_disambiguation* baseline, *PeerPressure* suffers from having an incorrect view on the distribution of configurations due to the ambiguity in Augeas library's output as discussed in Sec. 3.2.1. This problem is exacerbated when the misconfigured image has files that have substantially different parameter ordering compared to the files seen in the corpus. In these images, the parameters are represented by keys that use uncommon key indexing, making common configuration entries become outliers in the corpus and have high *PeerPressure* rankings.

5.2 Encore

Encore is a recently proposed tool that infers configuration constraints based on configurations collected from working systems [14]. In its original implementation, *Encore* uses the Augeas library to collect configurations and convert the collected information into key-value pairs. As we show in this paper, using Augeas only is not sufficient to discover configuration files in cloud instances, and the key-value pairs generated by Augeas are ambiguous, decreasing the robustness of corpus-based analysis. In this example, we demonstrate how *ConfEx* significantly increases the effectiveness of *Encore* on analyzing configurations in the cloud.

Encore has two steps: Parameter type inference and rule inference. Parameter type inference focuses on associating keys with configuration types such as integer, file path, or IP address. To reduce the amount of generated false types and rules, *Encore* focuses only on keys whose value show a certain level of *entropy* across the corpus, where entropy is a measure of diversity. Given a key-value pair, *Encore* first checks the syntax of the value to understand the type of the configuration key. For example, a value is identified as an IP address if it matches the regular expression `^\d{1,3}(\.\d{1,3}){3}$`. Next, *Encore* performs a type-specific semantic check to verify the type identification. For example, if a value is syntactically identified as a file path, the semantic step checks whether the given file path exists in the file system. If there is such a file exists, the type of the corresponding key is inferred as a file path.

In the rule inference step, *Encore* utilizes association rule learning to generate configuration constraints using type-specific rule templates. An example rule template is: "An entry should be equal to another entry of the same type". Such rule templates decrease the search space across the key-value pairs and reduces false rules. For each rule template, *Encore* generates all possible rules across the training corpus, then filters the generated rules based on *support* and *confidence*. Support is the fraction of images that have all the keys used in the proposed rule, and confidence is the fraction of the images where the rule is valid. *Encore* accepts a rule only if its support and confidence are at least 10% and 90%, respectively.

We use the configuration data extracted by *ConfEx* or one of the *Augeas* or *ConfEx_no_disambiguation* baselines (see Sec. 5.1) with *Encore* to infer configuration parameter types and configuration value constraints. Then, without injecting any misconfigurations, we identify the key-value pairs that violate the inferred parameter types and value constraints.

5.2.1 Encore Error Detection Results

Using the key-value pairs generated by our *ConfEx* framework, *Encore* detects 184 configuration errors. 181 of these errors are placeholder values that need to be replaced by external scripts. The values used in these files include `__PROXY_PASS__` and `{{(8*flavor['ram']/512)|int}}M`, which would lead to errors if these files are directly used. *Encore* also detects two file path errors where Windows-style paths are used in the configuration files of Docker Hub images. Another detected error is in the `/etc/passwd` file of an image, where the absolute path of a user's shell is set to `ata:ata:at`.

When using the *ConfEx_no_discovery* baseline, *Encore* finds only 99 of these errors due to the ambiguity in the key-value pairs. Using the *Augeas* baseline, which ignores files in non-standard paths both during training and testing, reduces the number of detected errors to 10.

6 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 [18], [42]. Due to their intrusiveness, however, instrumentation and trace analysis are often impractical on production workloads. Source code analysis [16], [34], [43] and natural language processing on application documentations [20], [44] have been used to infer configuration constraints before deployment. Configuration entries that do not comply with these constraints are then marked as errors.

Runtime-based configuration validation techniques [45], [46] have been used to check that the runtime behavior of an application matches an expected behavioral profile based on a valid configuration. Unexpected behavior is flagged as a possible misconfiguration. These techniques are largely application-specific and require either precise configuration of a monitoring daemon for the kinds of behavior being observed or modification of the application itself.

Application-agnostic statistical techniques [14], [15], [21] 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 do not address discovery or extraction for text-based configuration files.

In prior studies, the discovery and extraction of configurations in the cloud have been performed using several methods: parsing known configuration files with custom scripts (e.g., [20], [47]), crawling erroneous files from mailing lists and technical forums (e.g., [3]), parsing files located in default paths using configuration parsing libraries (e.g., [14]), and using standardized configuration stores such as the 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. Chef [12] and Ansible [13] have configuration editing capabilities that are restricted to search-

and-replace based on regular expressions, but they cannot extract configurations from text files. CFEngine [48] can parse standard file formats such as XML and JSON, but not application-specific configuration formats such as in httpd and Nginx. Puppet [29] and bcfg2 [30] can edit application-specific files by leveraging Augeas library [28]. As we show in this work, using the Augeas library alone is not sufficient for parameter extraction for robust configuration analysis.

Recently, Huang et al. proposed SAIC [49], 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. Xu et al. [50] have discussed the opportunities and challenges associated with mining container image repositories for software configurations, but have not devised a solution.

We have introduced a preliminary version of *ConfEx* in our recent work [51]. As opposed to this preliminary version, our framework in this paper has an active file discovery methodology, applies syntax check while identifying configuration files in addition to the vocabulary-based comparison, and records the order of parameters that appear in a configuration file as well as environmental configuration information. These improvements enable the detection of a wider set of misconfigurations. In addition to the above improvements, in this extended paper, we present a more detailed evaluation of *ConfEx* by using both *PeerPressure* and *Encore* as applied examples.

To the best of our knowledge, our configuration analytics framework, *ConfEx*, is the first to systematically discover and extract text-based software configurations in cloud instances with unlabeled content. In this way, *ConfEx* enables comprehensive analysis of software configurations in the cloud to help users validate their configurations and achieve robust operation.

7 Conclusion

Due to the increasing prevalence and severity of misconfigurations in cloud services, there is need for a cloud-based framework that can support analyzation and validation of software configurations. To enable automated configuration analysis, we have proposed *ConfEx*, a framework to discover and analyze text-based software configurations in multi-tenant cloud platforms. To identify configuration files in cloud instances with unlabeled content, *ConfEx* keeps track of configuration keywords such as parameter names and commands in text files and checks file syntax, achieving over 99% precision and 100% recall. To parse these files, *ConfEx* leverages a community-driven configuration parser, Augeas. It then disambiguates the parser output to obtain configuration key-value pairs that are consistent across different files and cloud instances. Our framework enables the use of existing configuration analysis tools, which are designed for key-value pairs, in the cloud, which we hope will lead to fewer configuration-related outages and a more secure and reliable cloud ecosystem.

Acknowledgement

This project has been partially funded by the IBM T.J. Watson Research Center.

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 Joint Meeting on Foundations of Software Engineering (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 International Conference on Autonomic Computing (ICAC)*, 2009, pp. 169–178.

[3] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 619–634.

[4] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 70:1–70:41, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2791577

[5] 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 (USITS)*, vol. 4, 2003, pp. 1–1.

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

[7] 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 ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 159–172.

[8] Y. Sverdlik, "Microsoft: misconfigured network device led to Azure outage," Jul 2012. [Online]. Available: https://www.datacenterdynamics.com/news/microsoft-misconfigured-network-device-led-to-azure-outage/

[9] K. Thomas, "Thanks, Amazon: The cloud crash reveals your importance," Apr 2011. [Online]. Available: https://www.pcworld.com/article/226033/thanks_amazon_for_making_possible_much_of_the_internet.html

[10] M. Welsh, "What I wish systems researchers would work on," May 2013. [Online]. Available: http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html

[11] B. Maurer, "Fail at scale," *Communications of the ACM*, vol. 58, no. 11, pp. 44–49, Oct. 2015.

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

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

[14] 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 687–700.

[15] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with PeerPressure," in *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, vol. 4, 2004, pp. 245–257.

[16] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *SOSP*, 2013, pp. 244–259.

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

[18] 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.

[19] F. Behrang, M. B. Cohen, and A. Orso, "Users beware: Preference inconsistencies ahead," in *ESEC/FSE*, 2015, pp. 295–306.

[20] 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," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1828–1839, Aug. 2015.

[21] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *Proceedings of the ACM on Programming Languages*, vol. 1, pp. 64:1–64:20, Oct. 2017.

[22] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.", 2016, ch. The Evolution of Automation at Google.

[23] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *IEEE Software*, vol. 33, no. 3, pp. 94–100, May 2016.

[24] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.

[25] R. Real and J. M. Vargas, "The probabilistic basis of jaccard's index of similarity," *Systematic biology*, vol. 45, no. 3, pp. 380–385, 1996.

[26] 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 (USENIXATC)*, 2011, pp. 28–28.

[27] 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.

[28] D. Lutterkort, "Augeas - a configuration API," *Red Hat Summit*, 2008.

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

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

[31] 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 International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2017, pp. 88–97.

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

[33] 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 ACM/IFIP/USENIX Middleware Conference: Industrial Track*, pp. 29–35.

[34] 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 *EASE*, 2017, pp. 196–201.

[35] M. Marks. (2016, Aug) Docker hub hits 5 billion pulls. https://blog.docker.com/2016/08/docker-hub-hits-5-billion-pulls/.

[36] T. Osbourn. (2011, Aug) 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/.

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

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

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

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

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

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

[43] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *ICSE*, 2014, pp. 140–151.

[44] D. Jin, M. B. Cohen, X. Qu, and B. Robinson, "Preffinder: Getting the right preference in configurable software systems," in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014, pp. 151–162.

[45] A. Jahanbanifar, F. Khendek, and M. Toeroe, "Partial validation of configurations at runtime," in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, April 2015, pp. 288–291.

[46] L. Akue, E. Lavinal, T. Desprats, and M. Sibilla, "Integrating an online configuration checker with existing management systems: Application to cim/wbem environments," in *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, Oct 2013, pp. 339–344.

[47] T. C. Chieu, M. Singh, C. Tang, M. Viswanathan, and A. Gupta, "Automation system for validation of configuration and security compliance in managed cloud services," in *2012 IEEE Ninth International Conference on e-Business Engineering*, Sep. 2012, pp. 285–291.

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

[49] 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

[50] T. Xu and D. Marinov, "Mining container image repositories for software configuration and beyond," *CoRR*, vol. abs/1802.03558, 2018. [Online]. Available: http://arxiv.org/abs/1802.03558

[51] O. Tuncer, N. Bila, S. Duri, C. Isci, and A. K. Coskun, "Confex: Towards automating software configuration analytics in the cloud," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, June 2018, pp. 30–33.

Ozan Tuncer is a Ph.D. candidate at the Department of Electrical and Computer Engineering of Boston University. He received his M.S. degree in Computer Engineering from Boston University, and his B.S. degree in Electrical and Electronics Engineering from the Middle East Technical University, Turkey. His research interests include data analytics for cloud system management, data center power and thermal management, and workload management for high performance computing.

Anthony Byrne is a Ph.D. student at the Department of Electrical and Computer Engineering of Boston University. He received his B.S. degree in Computer Engineering from Boston University. His current research interests include operating systems, embedded systems, and applications of machine learning in cloud systems.

Nilton Bila is a systems researcher at IBM. He builds large scale cloud platforms and addresses problems in systems configurations, cloud security, software dependability, and virtualization. He received his Ph.D. and M.Sc. degrees in Computer Science from the University of Toronto.

Sastry Duri is a senior software engineer at the IBM T. J. Watson Research Center in Yorktown Heights, New York. He earned a Ph.D. degree in computer science from the University of Illinois at Chicago. His professional interests include distributed computing systems, cloud monitoring and analytics, mobile commerce applications, and RFID based supply chains. In his spare time, Sastry coaches teams for FIRST Robotics competitions. In the past, he represented IBM in the industry standard group EPCglobal ALE Working Group, a subsidiary of the Uniform Code Council (UCC), and in OpenLS workgroup.

Canturk Isci is a principal researcher and master inventor in IBM T.J. Watson Research Center. He currently works on cloud monitoring, operational and security analytics. Prior to IBM Research, Canturk was a Senior Member of Technical Staff at VMware, where he worked on distributed resource and power management. Canturk has a B.S. in Electrical Engineering from Bilkent University, an M.Sc. with Distinction in VLSI System Design from University of Westminster, and a Ph.D. in Computer Engineering from Princeton University.

Ayse K. Coskun is an associate professor at the Department of Electrical and Computer Engineering, Boston University (BU). She was with Sun Microsystems (now Oracle), San Diego, prior to her current position at BU. Her research interests include energy-efficient computing, architectures built with emerging technologies, embedded systems, and large-scale systems analytics and management. She received the M.S. and Ph.D. degrees in Computer Science and Engineering from the University of California, San Diego. She currently serves as an associate editor of IEEE Transactions on CAD.