Learning Patterns in Configuration

Ranjita Bhagwan
Microsoft Research
bhagwan@microsoft.com

Sonu Mehta
Microsoft Research
someh@microsoft.com

Arjun Radhakrishna
Microsoft
arradha@microsoft.com

Sahil Garg
University of California, Berkeley
sahil_garg@berkeley.edu

***Abstract*—Large services depend on correct configuration to run efficiently and seamlessly. Checking such configuration for correctness is important because services use a large and continuously increasing number of configuration files and parameters. Yet, very few such tools exist because the permissible values for a configuration parameter are seldom specified or documented, existing at best as tribal knowledge among a few domain experts.**

In this paper, we address the problem of *configuration pattern mining*: learning configuration rules from examples. Using program synthesis and a novel string profiling algorithm, we show that we can use file contents and histories of commits to learn patterns in configuration. We have built a tool called ConfMiner that implements configuration pattern mining and have evaluated it on four large repositories containing configuration for a large-scale enterprise service. Our evaluation shows that ConfMiner learns a large variety of configuration rules with high precision and is very useful in flagging anomalous configuration.

I. Introduction

Configuration management is an integral part of the development and deployment of large services. These services depend heavily on correct configuration to run uninterrupted, be flexible to changing environments, and to scale seamlessly. This ubiquitous use of configuration in services poses several daunting challenges, one of which is to ensure that every configuration parameter is set to a suitable value. To make matters worse, the amount of configuration that a service needs to manage grows significantly with time as the service scales out and as developers add new features and capabilities. For instance, the Microsoft 365 email service [1] more than doubled the number of configuration files in just a period of six months.

Unlike code, for which compilers and static analysis tools catch several types of errors well before the developer commits their changes, very few tools exist to perform such checks on configuration. This is because rules governing which configuration values are appropriate for a particular configuration parameter are very specific to the scenario in which the service uses the configuration value. For instance, a configuration value may capture a timeout for a particular microservice which the microservice expects to be a few minutes. If a developer were to set this timeout value to a few milliseconds by mistake, while she would be syntactically correct, the microservice may fail because of a lower-bound check on the timeout value. Worse, if no such check exists, the microservice would start timing out much too soon in deployment and thereby cause service disruption. Such requirements of configuration correctness are seldom documented. Very often they are subtle, very specific to the context in which they are used, and difficult to catch through specification and hard-coded rule-based checking. Consequently, misconfigurations in large services occur much too often, cause build and test failures, and sometimes significant disruption and data breaches [2], [3], [4], [5]. For instance, in January 2020, Microsoft exposed 250 million customer records inadvertently because a database specified personally identifiable information (e.g. email addresses) in an anomalous format [6].

Towards addressing this, we observe a unique opportunity driven by two recent trends. First, modern services maintain configuration in files separate from code, such as yaml, json, or xml files. Engineers process configuration changes similar to code changes: they commit changes to the configuration through a version control system. We can therefore treat *configuration-as-data* by tapping into the version control system. Through commit logs and file histories, we have access to a rich history of configuration file snapshots and changes from which we can *learn* patterns in configuration values. Also, since configuration is gated by version control systems, we also have the opportunity to automate configuration checks at commit time and catch errors early, well before deployment.

Second, the field of program synthesis for data processing has seen rapid progress in recent years. Tools like FlashFill learn programs that capture patterns in values, structure and sequences and have been used successfully in various domains such as automated manipulation of tabular data [7], [8], [9], [10] and semi-structured data extraction [11], [12].

In this paper, we bring the ideas of program synthesis and configuration-as-data together to perform *configuration pattern mining*. We introduce a novel program synthesis-based string profiling procedure to learn regular expression based rules that capture patterns in configuration values. This procedure is based on the techniques presented in [13], but is significantly more efficient and robust to noise. For instance, given enough examples, the string profiling procedure can learn that a timeout value in a configuration file is always specified as a number followed by the character, “s" (the regular expression `[0-9]+s`). If a developer erroneously specifies a timeout value of “1ms” or a timeout value of “10” without specifying the units, it will not match the learned regular expression and hence, we can flag the mismatch as a potential configuration error.

We use two types of input data (or examples) to the string profiling algorithm to learn to different kinds of rules for configuration values. First, we learn *history-based rules* by using versions of the same configuration value in previous

commits and detect patterns in them. Second, we learn *file-based rules* that learn patterns in different configuration values within the same version of a single configuration file. We find that both rule types capture various configuration patterns and can be used to flag misconfigurations, auto-suggest correct values to developers as they edit configuration, and thereby contribute to improving configuration management.

We have built ConfMiner, a tool that uses the algorithms described here to implement configuration pattern mining. We have evaluated ConfMiner on 4 repositories that are used to manage configuration for a large-scale enterprise service used by hundreds of millions of users. Our results show that ConfMiner is very generally applicable and learns a wide variety of rules. We also observe that ConfMiner's rules catch many different kinds of misconfigurations very early in the development process. This can help accelerate the development cycle of services and improve service reliability.

This paper makes the following novel contributions.

- We introduce the problem of configuration pattern mining, i.e. learning patterns in configuration from examples that we get from file version histories and commit histories. To the best of our knowledge, this is the first work to target the problem of pattern mining in configuration values. We describe two types of configuration pattern mining: value-based and structure-based.
- We develop a novel program synthesis string profiling procedure and apply this to configuration-as-data to learn value-based rules pertaining to configuration. This procedure outperforms the state-of-the-art string profiling algorithm by 3.3X and is of independent interest outside the context of configuration mining.
- We have built a tool called ConfMiner that implements our algorithms for value-based rule mining. We have evaluated it on 4 configuration repositories for a large, popular service. Our results show that ConfMiner is indeed effective in learning patterns in configuration values.

The rest of the paper is organized as follows. Section II describes the problem of configuration pattern mining, outlines our solution, and describes the scope of this work. Section III describes the algorithm and system components in detail. Section IV presents our evaluation of ConfMiner and the string matching algorithm. Section V discusses threats to validity. Section VI discusses related literature and section VII provides a conclusion to the paper.

II. Problem Definition and Scope

In this section, we first explain configuration pattern mining with an example configuration file. Next, we provide an overview of how ConfMiner performs configuration pattern mining and how applications can use these patterns. Finally, we discuss the scope of our work and bring out its inherent limitations.

A. Configuration Pattern Mining

Any configuration file, be it in json, xml, yaml, or any other format, can be expressed as a hierarchical tree structure. Each node is a key-value pair, where the key is the configuration parameter and the value is the value that the configuration parameter is set to. An edge connecting two nodes captures a parent-child relationship between configuration parameters. Given this, we envision two types of configuration pattern mining: value-based and structure-based.

```
# Google App Engine application config file

application: foobar
version: 5
runtime: php55

# Manifest files

- url: /(.+\(appcache))
  static_files: \1
  upload: static/(.+\(appcache))
  mime_type: text/cache-manifest
  expiration: "0s"

- url: /(.+\(webapp))
  static_files: \1
  upload: (.+\(webapp))
  mime_type: app/x-web-app-manifest+json
  expiration: "5s"

# CSS, Javascript, text etc

- url: /(.+\(css|js|xml|txt))
  static_files: \1
  upload: (.+\(css|js|xml|txt))
  expiration: "5m"

# HTML pages

- url: /(.+\(html))
  static_files: \1
  upload: (.+\(html))
  expiration: "10m"
```

Fig. 1. Example configuration file derived from Google AppEngine's `app.yaml`. We have modified this file for illustrative purposes here.

1) Value-based pattern mining: This is the process of learning patterns in the values of a configuration parameter. We use regular expression learning to do this. Consider the `app.yaml` configuration file in Figure 1 which is an abridged example of a Google App Engine's configuration file [14]. The parameter `expiration` (highlighted in blue) is set four times in different sections of the file, and all values follow a certain pattern: a number followed by an 's' (for seconds) or an 'm' (for minutes). From this, we can learn patterns for the value (or a rule) as the regular expression `[0-9]+[s|m]`. Since we learn this rule based on the contents of the file alone, we call this a *file-based rule*. Now, notice the parameter `version` (highlighted in red) on the second line. This is specified only once in the file as `5`, but say previous values of this parameter in earlier revisions of this file were `1`, `2`, `3` and `4`. From all these values, we can learn that this parameter is always a number, i.e., follows the pattern specified as the regular

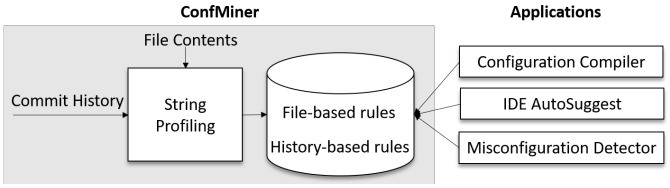

Fig. 2. ConfMiner System Overview

expression `[0-9]+`. We can learn this rule from a history of commits to this specific configuration parameter. We call this a *history-based rule*.

Notice that the generalizations these rules provide are not unique. For any given set of values, we can learn multiple regular expressions. For instance, in the case of the `expiration` parameter, we could learn a more general expression such as `[0-9]+[a-z]` which allows any letter, not just 's' or 'm'. Or, we could learn a more specific regular expression e.g. `[0|5|10][s|m]` which only allows numbers `0`, `5` and `10`.

2) Structure-based pattern mining: Configuration files have a rigid structure. In Figure 1, the file specifies a list of four elements: each element is prefixed by a '-'. Each list element has a specified set of configuration parameters. All four elements have parameter `expiration` specified (in blue), whereas, only two out of four elements have `mime_type` specified (in green). From this, we could infer that `expiration` is potentially a mandatory parameter whereas `mime_type` is not. Another form of structural pattern is that there may be an implicit ordering requirement of parameters. Certain parameters may have to be specified before others. A common example of this is firewall configurations, where the configured rules are applied in order to determine whether a network connection should be allowed or denied. Each rule overrides all other rules that appear after it in the ordering. This paper concentrates on value-based mining; we leave work on structure-based pattern mining algorithms to future work.

B. System Overview

ConfMiner's goal is to learn patterns for configuration that can be used by several applications to detect misconfiguration, suggest changes, and build automatic checkers. Figure 2 shows an overview of the system. First, ConfMiner runs a string profiling procedure on a specific file's contents to generate file-based rules for every configuration parameter that has a large enough number of example values in the file (e.g., the parameter `expiration` in Figure 2). Next, ConfMiner uses commit histories for a specific configuration parameter to learn history-based rules in a similar manner. All learned rules, their confidence and support are stored in a database. The rule-learning algorithm runs periodically on file contents and histories, e.g. in our deployment, ConfMiner learns rules once a day.

Applications query ConfMiner through a simple interface which, given a parameter and its value, return all rules that match that value. Applications can use this interface in multiple ways. For instance, an auto-checker can, at review time, post an automated comment if a commit to a particular parameter does not match any of the learned rules. This application is similar to commit recommendation systems such as Rex [15]. An IDE can also use the rules to suggest changes to configuration parameters as the developer starts to type in the change. This scenario is similar to IntelliSense in VisualStudio [16] or Content Assist in Eclipse [17]. A third application is to enable building automated configuration compilers and verifiers which can run along with code compilers, perhaps as plugins, to generate warnings, etc. This usage is similar to StyleCop [18].

C. Scope

We believe that configuration pattern mining is a vast subject which needs to be tackled through a significant body of research. In this paper, we concentrate primarily on value-based patterns that can be captured through regular expressions. Here is what the paper *does not* try to achieve.

1) Building a sound, complete system: ConfMiner is fundamentally a best-effort system, and since it learns from example values, it cannot be sound or complete. Notice that given a set of sample configuration values, our example in Section II-A showed that ConfMiner could learn more than one valid regular expression given a set of inputs. An application's efficacy will vary depending upon how specific the rules are. The more specific the rules, the more strict an application will be in enforcing them. The more generic the rules are, the application will be less strict but may miss out on valid misconfigurations. To handle both scenarios, ConfMiner learns a combination of rules that are very specific as well as very generic. An application can then use thresholds to select more specific or more general rules depending on its tolerance to false-positives and false-negatives.

2) Going beyond regular expressions: Our work concentrates only on patterns that can be captured via regular expressions. Not all value-based rules can be captured by regular expressions. For instance, values of different parameters could be correlated. Or, in certain cases, the value of a particular parameter may change in a very specific way over time, such as the `version` parameter in Section II-A goes up by 1 every time it is changed. We leave learning such rules and properties to future work.

3) Designing applications: Our goal is to design a pattern mining algorithm for configuration. While we do not concentrate on building applications that can use the mined configuration patterns, we have emulated an automated comment generator which, in real time, flags many misconfigurations made by developers in deployment. Section IV describes our evaluation and application emulation in detail.

III. ConfMiner Design

In this section, we first give some background on string profiling and describe our program synthesis-based string profiling algorithm, while providing details on the configuration-specific parameters. Next, we describe the data generation engine and the rule-learning engine. Finally, we describe the

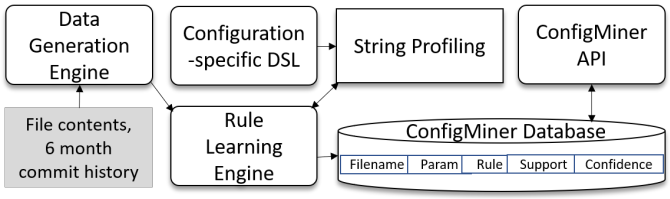

Fig. 3. ConfMiner components.

API that ConfMiner exposes and that applications can use. Figure 3 gives a summary of ConfMiner's components and how they interact.

A. Profiling Configuration Values

The string profiling problem. First, we address the problem of characterizing all the values a configuration parameter may take. Given the known set $\mathcal{S}$ of string values for a configuration parameter, the *string profiling problem* is to produce a set of disjoint regular expressions $\mathcal{R}$ and a set of outliers $\mathcal{O} \subseteq \mathcal{S}$ such that $\forall s \in \mathcal{S}.s \in \mathcal{O} \vee \exists r \in \mathcal{R}.s \in r$. That is, every string in $\mathcal{S}$ either matches one of the regular expressions in $\mathcal{R}$ or is in the outlier set $\mathcal{O}$. Here, we use the notation $s \in r$ to represent that a string s matches a regular expression r.

The set of regular expressions $\mathcal{R}$ is called the *profile* of $\mathcal{S}$. Intuitively, each $r \in \mathcal{R}$ defines a cluster of similar strings in $\mathcal{S}$. We will use the set of known values (obtained from the file or the history) for a configuration parameter to learn a profile $\mathcal{R}$ and use the profile as a specification for any new values of the configuration parameter.

The correctness requirements of the string profiling problem are rather easy to satisfy. The profile consisting the single regular expression `.*`, i.e., the expression to match all strings, is always a valid solution. To obtain useful solutions, we need to define an optimality criterion. We do not provide the full formal details of the criterion, but instead refer the reader to [13]. Intuitively, an optimal profile is one such that $\sum_{r \in \mathcal{R}} \mathsf{Score}(r)$ is minimized, where Score is a custom defined ranking function. We use the same Score function used in [13]—in particular, in the Microsoft PROgram Synthesis using Examples SDK (PROSE) [19] implementation of [13]. This version of Score was tuned using over 100 sample datasets across different application and data domains beyond just configuration mining.

The function Score assigns scores based on two factors: *specificity* and *simplicity*. For example, regular expressions that use the character class `.`, i.e., class that matches all characters are given a high score, while expressions using long constant strings are given a lower score. On the other hand, the expression `[0-9]*` would be given a lower score than `[0-9]{2,9}` despite the latter being more specific: the former is a simpler pattern.

We extend the optimality criterion to a noisy setting as: $\sum_{r \in \mathcal{R}} \mathsf{Score}(r) + o \cdot |\mathcal{O}|$ where $o \in \mathbb{R}^+$ is the *outlier penalty*. We explicitly penalize outliers using the parameter o, and tuning the value of o controls the balance between patterns and outliers.

Example III.1. *Consider the following set of values that the* `ResourcePath` *configuration parameter takes:* $\mathcal{S} = \{$

```
"resource/2020-08-26/first.xml",
"resource/2001-11-05/second.xml",
...,
"resource/1992-03-15/third.xml",
"deployed/main.xml",
"deployed/secondary.xml",
...,
"deployed/tertiary.xml",
"test_resource.xml" }
```

Here, the values fall into the following categories: (a) Values that match the regular expression r_1 = `resource/[0-9]{4}-[0-9]{2}-[0-9]{2}/[a-zA-Z]*[.]xml`*, (b) Values that match the regular expression* r_2 = `deployed/[a-zA-Z]*[.]xml`*, and (c) the outlier value* `test_resource.xml`*.*

Ideally, a string profiling procedure will characterize the patterns in $\mathcal{S}$ *with the regular expressions* $\mathcal{R} = \{r_1, r_2\}$ *and the outlier set* $\mathcal{O} = \{$`text_resource.xml`$\}$*. This profile would signify that any new values for the configuration parameter should match either* r_1 *or* r_2*.*

The Stochastic String Profiling Procedure. Algorithm 1 describes a stochastic algorithm for the string profiling problem. At its core, the algorithm uses the LearnRegex to learn a single regular expression r from a set of *generators* $G \subseteq \mathcal{S}$. In general, the set G is small, between $2-5$ strings. We use the LearnRegex procedure from [13], as implemented in the Microsoft PROgram Synthesis using Examples SDK (PROSE) [19].

Algorithm 1 Stochastic string profiling algorithm

Require: Set of strings $\mathcal{S}$
Require: Ranking score Score
Ensure: Regular expressions $\mathcal{R}$ and outliers $\mathcal{O} \subseteq \mathcal{S}$
1: $\mathsf{Clusters} \leftarrow \emptyset$
2: **while** * **do**
3: $G \leftarrow \mathsf{Sample}(\mathcal{S}, \mathsf{Clusters})$
4: $r \leftarrow \mathsf{LearnRegex}(G)$
5: $\mathsf{Clusters} \leftarrow \mathsf{Clusters} \cup \{G \mapsto r\}$
6: **end while**
7: $\mathcal{R} \leftarrow \mathsf{ApproxExactSetCover}(\mathsf{Clusters}, \mathsf{Score})$
8: $\mathcal{O} \leftarrow \{s \in \mathcal{S} \mid \forall r \in \mathcal{R}.s \notin r\}$
9: **return** $(\mathcal{R}, \mathcal{O})$

Given a set of strings $\mathcal{S}$, Algorithm 1 maintains a dictionary Clusters that maps subsets G of $\mathcal{S}$ to the regular expression $r = \mathsf{LearnRegex}(G)$. Each item $G \mapsto r$ in Clusters is a potential cluster in the learned profile, representing the strings $\{s \in \mathcal{S} \mid s \in r\}$. The procedure proceeds through the following stages:

Generate. We repeatedly sample small subsets (size $2-4$) of $\mathcal{S}$ and learn a regular expression using the LearnRegex procedure. During sampling, we do not construct G by uniformly

sampling from $\mathcal{S}$. Instead, we obtain G as follows: we start with an empty G and extend G with one of the following randomly chosen options 2–4 times:

- String $s \in \mathcal{S}$ that does not belong to any cluster,
- All the generators G of a cluster $G \mapsto r$ in Clusters, and
- String s where $s \in \mathcal{S}$ does belong to a cluster in Clusters,

This biased sampling attempts to achieve one of the following: (a) construct a new cluster out of the strings that do not belong to any cluster, (b) merge or extend existing clusters to form a larger one, and (c) construct new clusters independently of existing ones. Ideally, the sample-and-learn loop is run until all patterns in the desired profile are added to the Clusters collection. However, we do not know the desired patterns: in practice, we sample until no new regular expressions have been added to Clusters for 10 iterations.

Select. Now, given a set of candidate clusters and the Score function, we use an approximation algorithm for the minimal exact set cover to pick a near optimal subset of clusters. Given a set X and a set of its subsets $Y = \{X_1, \ldots, X_n\}$ with a cost function mapping X_i to reals, the exact set cover problem asks to choose a subset of $Y' \subseteq Y$ such that each $X_i, X_j \in Y'$ are disjoint and $\bigcup_{X_i \in Y'} X_i = X$. Of all such possible Y' we prefer the one with the minimal total cost. In our setting, (a) X is the set of all strings $\mathcal{S}$, (b) Y contains the set of candidate clusters Clusters, and (c) the cost function is Score.

The approximation algorithm follows standard greedy set cover algorithms: it maintains a partial solution, in this case, a set of regular expressions $\{r_1, r_2, \ldots, r_k\}$. In each iteration, the cluster $G \mapsto r$ which maximizes $|\{s \in \mathcal{S} \mid s \in r \wedge \forall i. s \notin r_i\}|/\mathsf{Score}(r)$ is added to the partial solution, and all clusters which intersect with r_i are discarded from Clusters.

However, we do not proceed until all strings in $\mathcal{S}$ are covered in the solution. Instead, we stop adding to the partial solution when the value $|\{s \in \mathcal{S} \mid s \in r \wedge \forall i. s \notin r_i\}|/\mathsf{Score}(r)$ drops below the outlier penalty o^{-1} for all r in Clusters. The strings in $\mathcal{S}$ that are not matched by any r_i in the solution are deemed outliers $\mathcal{O}$. Our implementation additionally returns the confidence and support for each pattern, which are defined as the fraction and the number of strings, respectively, that are matched by the corresponding regular expression.

It should be noted here that the fact that the procedure ignores a small fraction of outliers is particularly of importance to us, since this ensures that any rare examples of that configuration parameter which might be existing or historical misconfigurations are not matched in the learnt profile.

Example III.2. *Consider the set of strings $\mathcal{S}$ from Example III.1. In the first phase of the algorithm, we sample subsets of $\mathcal{S}$ and learn regular expressions from these samples. Here, we have 4 separate cases:*
(a) The sample only contains strings of form `"resource/{date}/{file_name}.xml` *In this case,* LearnRegex *returns* r_1.
(b) The sample only contains strings of form `deployed/{file_name}.xml`. *In this case,* LearnRegex *returns* r_2.
(c) Sample contains strings of both forms. Here, r_3 = `[a-zA-Z/]+/[a-zA-Z]*[.]xml` *is returned.*
(d) The sample contains the outlier string `test\_resource.xml`. *Here,* r_4 = `[a-zA-Z/_]*[.]xml` *is returned.*
By design, the ranking score Score *function produces scores with* $\mathsf{Score}(r_1), \mathsf{Score}(r_2) < \mathsf{Score}(r_3) < \mathsf{Score}(r_4)$, *by the principle of specificity.*

During the second phase, selection, the clusters are chosen using the greed heuristic, with r_1 and r_2 picked in sequence. For the rest of the clusters, the normalized score is less than the outlier penalty o^{-1}. Hence, the string `test_resource.xml` *is deemed an outlier. Note that the selection of regular expressions depends heavily on the* Score *function. With different* Score *functions, there are cases where the preferred cluster may be r_3 or r_4.*

Comparison to [13] We do not go into a full comparison of Algorithm 1 with [13] as it is not related to the main thesis of this paper. Summarizing the performance aspect of the comparison, Algorithm 1 is $3.3X$ faster than [13] on the set of benchmarks from [13] (see Figure 4). This performance improvement can be attributed to avoiding the expensive agglomerative hierarchical clustering (AHC) based approximation, which is $O(n^2)$ in the number of input strings. The time taken by Algorithm 1 is dominated by the sample

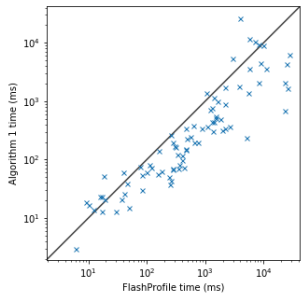

Fig. 4. Performance comparison of Algorithm 1 vs. FlashProfile

phase. The number of sampling iterations to produce good profiles depends on the number of patterns in the desired profile as opposed to the number of input strings. As the number of patterns in the profile is significantly smaller than the number of input strings, Algorithm 1 is often more efficient than AHC.

Further, Algorithm 1 often produces profiles of a higher quality than [13]. In the string profiling setting, AHC is sensitive to incorrect decisions in a manner that is not possible for numerical data. AHC proceeds by initially considering each string to be a cluster by itself, and then iteratively merging the two clusters that are the closest to each other. Here, the notion of distance is given by the Score value of the regular expression generated by the strings in the cluster. One incorrect merging decision, often due to outliers or similar strings of different patterns, has a cascading effect and may produce undesirable profiles.

Example III.3. *Consider a set of strings representing dates*

of the forms `14 January 2020` *and* `23-Feb-2020`*. Suppose the set contains the strings* `03-May-1992` *and* `03 May 1992`*. It is likely that the first merge performed by AHC will group these strings together to obtain the regular expression* `03[ -]May[ -]1992`*. This is because the resulting regular expression is very specific and will have a low* Score *value.*

After this point, all subsequent clusters will either include both these strings, or exclude both of them. Hence, we can never obtain the desired regular expressions `[0-9]{2}-[A-Z][a-z]{2}-[0-9]{4}` *and* `[0-9]{2} [A-Z][a-z]* [0-9]{4}`*. Instead, AHC returns a profile that has the single regular expression* `[0-9]{2}[ -][A-Z][a-z]*[ -][0-9]{4}` *which mixes the two formats. Similarly, in the presence of outliers, one incorrect decision merging an outlier with a non-outlier cluster will cause significant degradation of the final results. Algorithm 1 avoids these issues—even if the sampling groups strings incorrectly, the clusters that arise from these groups will be safely ignored during the selection phase.*

B. Token DSL for Configuration Mining.

We use the PROSE SDK implementation of the LearnRegex procedure. However, this procedure had to be customized to the context of configuration mining. During the learning process, LearnRegex constructs regular expressions using a domain specific language DSL of generic tokens such as `[0-9]` for digits and `[A-Z]` for uppercase letters. ConfMiner has had to modify the DSL to make it more configuration-specific. Consider the following example scenarios.

We found that configurations often capture names, such as file names, test names and firewall rule names. Names can have an arbitrary number of camel-cased terms. For instance, a parameter `testname` captures the names of tests to run on a particular code-base. Say ConfMiner's data collection found three unique values for it: `("testComponent","testData","testSystem")`. Given the generic DSL, ConfMiner would learn the regular expression `test[A-Z][a-z]+` which allows only one camel-cased term to follow `test`. Hence if an application queries ConfMiner with the value `"testAppData"` for this parameter, the regular expression will not match it since it has two camel-cased terms following `test`. To accommodate this scenario, the configuration-specific DSL ignores the token `[A-Z]` and hence ConfMiner learns the regular expression `test[A-Za-z]+` which is much more general and allows an arbitrary number of camel-cased terms in the name.

The configuration-specific DSL also includes new tokens. For instance, we found that values often capture lists of arbitrary sizes, where a delimiter such as a comma separates the list elements. To allow for lists of arbitrary size, the configuration-specific DSL includes tokens such as `(\w+,)+` which covers comma-separated lists or arbitrary size. If the DSL does not include such list-specific patterns, ConfMiner learns regular expressions only towards a specific number of elements in the list which again causes incorrect matching behavior in a large number of cases.

C. Data Generation Engine

We now describe the data generation process for file-based and history-based rules.

1) File-based: Data generation to learn file-based rules is triggered every time a configuration file is changed. ConfMiner's data generation engine parses the configuration file using format-specific parsers. Currently, our implementation supports 11 different file types including xml, json, yaml and ini. Each parser gives us a tree object that captures all the configuration in a structured format. From this, ConfMiner extracts tuples of the form `(file_name, param_name, list_of_values)`. A parameter name, such as `expiration` in Figure 1, can exist under different parent configurations. We made the choice to ignore the ancestry of each parameter and, as long as their names are identical, ConfMiner combines all values of parameters within the same tuple and into one list of values. Hence, for the mentioned example, ConfMiner generates the tuple `(app.yaml, expiration, ["0s", "5s", "5m", "10m"])`. This increases the number of values for each parameter and more data allows for rules with higher confidence and support from the string profiling algorithm.

2) History-based: Data generation for history-based rules is triggered every time a commit changes an existing parameter in a configuration file. For every file commit, ConfMiner runs a differential analysis on the file to detect which particular configuration parameter has changed. A textual difference (which version control systems readily provide) does not suffice because it is possible that the value of a parameter spans multiple lines, and if only some of those lines are changed, one cannot tell what the changed parameter is. ConfMiner therefore performs the difference at a syntactic level. To do this, it uses configuration parsers to learn the tree object for the old version of the file, does the same for the new version of the file, and compares them using heuristic approaches. From these comparisons, it finds a) the changed configuration parameter and b) the new value it is set to. From this, ConfMiner creates tuples of the form `file_name, param_name, list_of_values)`.
In addition, ConfMiner also generates new tuples that combine data across files if the configuration parameter name (`param_name`) is the same. This aggregated data is particularly useful towards learning rules that govern generic datatypes such as IP addresses, and DLL version numbers which could have the same format across different files. Section IV-C shows several examples of file-based rules and history-based rules that ConfMiner learned in deployment. This includes examples of generic patterns that exist across files as well.

Depending on file format, configurations could have slightly varying structure. For instance, in the xml format, a configuration parameter, apart from having a value, could have *attributes* which themselves have set values. ConfMiner accom-

modates all these specific details for different formats. More details on the implementation are provided in Section IV-A.

D. Rule-Learning Engine

Once ConfMiner generates data for file-based and history-based rules separately, it uses the string profiling algorithm which returns a list of regular expressions with the confidence and support for each. While rules with higher confidence and support are indicative of "well-behaved" values of a configuration parameter, rules with very low confidence or support may be equally important and useful. At first glance, this appears counter-intuitive. But several configuration parameters, in reality, have very varied patterns like the `url` parameter in Figure 1. Learning one regular expression that captures all these parameters would make the regular expression too generic, albeit with high confidence and support. We have found that it is indeed better to learn a small number of regular expressions that capture the whole set of values, with each regular expression having relatively low confidence and support. We use the Score parameter in the string profiling algorithm to strike this balance and set a very low threshold on the confidence and support of the rules learned. The confidence threshold is set to 0.03, i.e., if a cluster contains 3% of known file-based or history-based values for a configuration parameter, the corresponding rule is considered valid. For rules with confidence lower than this threshold, the corresponding values are considered accidental or outliers, i.e., even if a new configuration value matches this rule, it is considered potentially incorrect and flagged.

Currently, ConfMiner's rule-learning engine is triggered once a day. Using all commits within that day, it learns history-based rules using a commit history of 6 months. It learns file-based rules for any configuration file that has been changed on that day. Finally, ConfMiner stores all learned rules in a central database, indexed by file name and/or parameter name.

E. ConfMiner API

The above sections have described the process by which ConfMiner learns regular expressions given input examples. We now describe the API using which applications can use ConfMiner. The primary call that ConfMiner supports is `FindMatches(file_name, param_name, value)`. The call returns a list of all file-based and history-based rules that the value matches along with the confidence and support for each rule. The application can then further filter the set of rules based on its own confidence and support requirements. If no matching rule is found, the function returns a null value. ConfMiner also returns matches using generic history-based rules, which hold across different file names. Again, the application can decide to keep these rules or eliminate them.

IV. Evaluation

We first describe details of the ConfMiner implementation and describe our deployment of ConfMiner on 4 repositories of Microsoft 365, a large-scale widely used enterprise service. We then evaluate the string profiling algorithm using real data from these repositories. We emulate an automated misconfiguration checker and show that ConfMiner rules flag several misconfigurations as and when developers commit them. Finally, we examine 64 real-world configuration issues as reported in the Ctest dataset [20] to determine how often configuration pattern mining can help find real misconfigurations.

A. Implementation

ConfMiner is implemented using 4760 lines of C# code and works with Git [21]. The data generation engine interfaces with both Github [22] and Azure DevOps [23]. It supports a total of 11 file types that typically store configuration, including xml, json, yaml, csproj, config and ini. For each format, the data generation engine implements parsers which first translate the file contents into the xml format. ConfMiner then inputs the xml for the old version and the new version to a *difference module* which implements the differential syntactic analysis required to learn history-based rules. The difference module is built using the `XmlDiffAndPatch` [24] library. The ConfMiner API is implemented using approximately 1500 lines of C# code. In our Azure DevOps implementation, we use service hooks [25] to capture commits to configuration files as and when they happen.

B. Deployment

We have deployed ConfMiner on four repositories belonging to a large continuously deployed service within our enterprise, as shown in Table I. R1 contains both code and configuration of core features in the service. R2 contains information regarding physical configuration, such as datacenter-specific configuration and network-specific configuration. R3 contains code and configuration related to the DevOps environment i.e. build, test and deployment pipeline for the service. Finally, R4 contains code and configuration for all applications built on top of the core features that the service provides.

Most configuration for these repositories sits in xml, json and ini files. The "config changes" column tells us the number of configuration parameters that have been changed in the last six months. This is as high as 23,521 in the case of R4. The rule-learning engine uses these changes to learn history-based rules. Increasing history beyond six months makes the computation of rules slow and also biases the rules towards older data that may not be relevant. The "file changes" column gives the number of configuration files changed in the last six months. This is much lower than the configuration changes because every file captures multiple configuration parameters. Finally, the "history-based rules" column and the "file-based rules" column show the number of rules ConfMiner learns in these two categories using six months of data.

C. Example Rules

Table II gives some example rules that ConfMiner has learned in deployment. Both file-based and history-based techniques learned similar rules. As can be seen the support i.e. the number of input examples that match each learnt rule can vary widely, sometimes reaching a few thousand. Also,

TABLE I
DETAILS OF REPOSITORIES THAT CONFMINER IS DEPLOYED ON. FILE CHANGES AND CONFIGURATION FILE COMMITS ARE FOR THE LAST SIX MONTHS.

Repository	Config file counts			Config changes	History-based rules	File changes	File-based rules
	json	xml	ini				
R1	4635	55815	5199	21627	13358	976	36089
R2	1905	3104	114	18461	14313	634	17261
R3	598	10150	121	2130	8173	337	5816
R4	6769	2008	1967	23521	13113	661	15443

TABLE II
EXAMPLE RULES LEARNED.

No.	File type	Config name	Rule	Examples	Support
1	xml	`ServerName`	`SRDC-NLB-0[0-9]A`	“`SRDC-NLB-01A`" “`SRDC-NLB-03A`"	34
2	config	`Version`	`[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+`	“`19.12.12.151005`" “`3.7.25810.101`"	466
3	xml	`Runtime`	`00:00:00\.[0-9]7`	“`00:00:00.2178760`" “`00:00:00.3280206`"	7219
4	csproj	`Include`	`tbuild\_DB[0-9a-fA-F]+\.xml`	“`tbuild_DB3.xml`" “`tbuild_DB9f.xml`"	17
5	xml	`Duration`	`00:00:[0-9]{2}`	`"00:00:24"` `"00:00:01"`	10
6	-	`TimeServers`	`10\.22\.5\.134,10\.89\.0\.13`	-	1043
7	json	`Normal`	`0x[0-9a-fA-F]{6}`	“`0x212838`" “`0x7C68B2`"	33
8	json	`TestProfiles`	`[A-Z][a-z]+,Search,[A-Z][a-z]+`	“`Suggestions,Search,Teams`" “`Recommendations,Search,Suggestions`"	45
9	ini	`OrderBy`	`UserRelation/[a-zA-Z]+[\s]desc`	“`UserRelation/TrendingWeight desc`" “`UserRelation/LastAccessTime desc`"	14
10	ini	`ItemName`	`AutoSuggest\.[0-9a-zA-Z]+`	“`AutoSuggest.L0RankerControlCsgIndex`" “`AutoSuggest.L1v3RankerCsgIndex`"	108

ConfMiner learns many rules across all major configuration file types such as `xml`, `json` and `csproj`. Rules learned many different kinds of patterns. Row 1 shows how ConfMiner learns formats in machine names. Row 2 demonstrates that version numbers in a particular configuration file consist of four numbers separated by a “.”. Rows 3 and 5 capture two different time formats. Row 4 shows an example pattern in included file-names in a project file. Row 6 is an example of an aggregated history-based rule across many files. It learns two very specific IP addresses. Row 7 infers a 7-digit hexadecimal pattern. Row 8 learns a list of strings with a specific pattern, i.e. the word “Search" is always second in the list. Finally Row 9 and row 10 show miscellaneous examples of configuration value patterns that specify an ordering relationship and an auto-suggest algorithm.

As these examples show, patterns of very different types exist across various configuration values that have a wide array of semantics. Using a generic program synthesis framework enables ConfMiner to be relevant in a large number of scenarios which are very different from each other. Moreover, this varied set of rules shows that specifying such rules manually is a formidable task which cannot be achieved at scale without developers making a huge investment in time and effort.

D. Precision

We perform an online evaluation of ConfMiner on these four repositories. Notice that since ConfMiner is actually deployed on these repositories, evaluation is more realistic than a standard train-test split based evaluation. When a developer completes a pull-request that changes a configuration file, ConfMiner first determines which configuration parameter is changed, and what value it is changed to. To do this, ConfMiner uses the same difference algorithm as used to generate the data to learn history-based rules. Then, for each changed parameter, we call the `FindMatches` function provided by the ConfMiner API. ConfMiner checks if the change matches any of the rules learned in the previous day (ConfMiner's rule-learning engine runs once a day.). If it does, the change is labeled a *true-positive (TP)*, i.e. ConfMiner has indeed learned a rule that the new value matches. If no match is found, the change is labeled a *false-positive (FP)*, i.e. ConfMiner was not able to learn a rule that matches the new value. The *precision* of ConfMiner is calculated as TP/(TP+FP). Note that we assume that if a developer is completing a pull-request with a changed configuration file, it must be correct. It is of course possible that the value is wrong, and the developer corrects it later. We make the reasonable assumption that such situations arise only rarely.

Table III shows the total precision, precision due to file-based rules, and precision due to history-based rules for all configuration changes made from 18th of July 2020 to date. Total precision lies between 0.7 and 0.85. In general, history-based learning shows overall better precision than file-based learning, with the value being as high as 0.92. The slightly lower precision of file-based rules is because a lot of these rules learn patterns in file paths and file names. Thus when developers introduce a new file path, or change the format

TABLE III
OVERALL PRECISION, FILE-BASED PRECISION AND HISTORY-BASED PRECISION THAT CONFMINER ACHIEVES.

Repository	Total			File-based			History-based		
	TP	FP	Precision	TP	FP	Precision	TP	FP	Precision
R1	279	105	0.73	148	82	0.64	129	21	0.86
R2	372	157	0.70	257	140	0.65	112	17	0.87
R3	60	20	0.75	36	15	0.71	24	5	0.83
R4	639	111	0.85	227	73	0.76	408	37	0.92

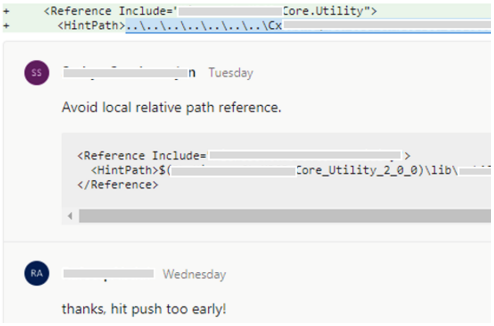

Fig. 5. Example manual comment that can be replaced by ConfMiner. All de-anonymizing information is elided.

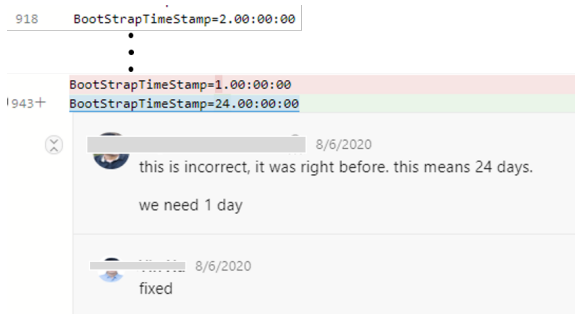

Fig. 6. ConfMiner flagged this since the timestamp did not match the rule.

of a file name slightly, this is recorded as a false-positive. Sometimes, these new file paths or name format changes are required and therefore our rules are indeed wrong. However, very often we found that our rules, though marked as false-positives, were indeed valid because they required the format of the file path or name to be fully consistent with previously seen examples, and the developer had committed a path or name in a slightly different format. We explain one such example in detail in Section IV-E3. Such rules, though they do not flag misconfigurations, do point to style defects addressing which can improve hygiene and readability of configuration.

E. Application Emulation

Section IV-D gives us confidence in the inherent ability of ConfMiner to learn rules with high precision. Now, we ask how useful these rules are in flagging misconfiguration. For this, we have built a misconfiguration detection application on top of ConfMiner which tracks every commit to a configuration file. If the commit does not match any rules, a flag is raised that this is a potential misconfiguration. The flag is silent, i.e. the developer is not informed of the flag. Hence, we call this an *emulation* of the application. If, before completing the pull-request, the developer changes the configuration value again so that it now matches a learned rule, it indicates that the previous value was indeed incorrect and the flag that our application raised was valid. We now present a few example misconfigurations that our emulation flagged. We have used redacted screenshots to keep them anonymous.

1) Path-based misconfiguration: We notice that configuration values that hold file names and file paths are often misconfigured with relative paths rather than absolute paths. ConfMiner has been very effective at flagging this. Figure 5 shows an example pull-request that ConfMiner flagged. A reviewer manually calls out this error for correction through the comment, thereby confirming that the flag was indeed valid. Given that ConfMiner flagged this error by the developer, we believe that the comments on such errors, which are currently entirely manual, can be automated. Moreover, since ConfMiner takes at most a few seconds to flag such errors, i.e. the time it takes to call the ConfMiner API, it can flag such errors much faster than the manual review process.

2) Numerical misconfigurations: In Figure 6, we see that in line 943, the developer changed a value for `BootstrapTimeStamp` from `1.00:00:00` to `24.00:00:00`. However, in multiple other sections of the file, for instance in line 918, the same configuration has been set to `2.00:00:00`. A file-based rule therefore learned that this configuration should be set such that the first number that appears is a single digit, and not two digits, as in 24. Notice that the reviewer also made this comment, albeit with much more semantics, saying that the developer has set the number of days as 24, and not the number of hours. This example shows that ConfMiner, through its simple regular express learning, can sometimes flag subtle misconfigurations even without understanding the semantics. The challenge though, if we are to automate comments based on this, is to make automated comments capture semantics like the reviewer has in this example. We leave this to future work.

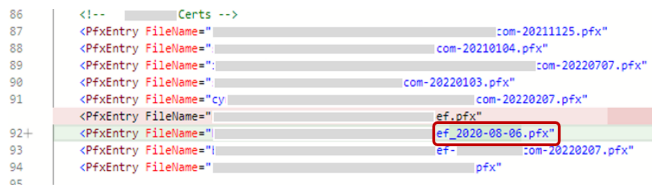

Fig. 7. ConfMiner flagged this change because of an anomalous file name.

3) Style recommendations: ConfMiner flags several style-related recommendations as well. Figure 7 shows an example

where ConfMiner learns a specific file format: a string-based name, followed by a hyphen and a date. The shown commit on line 92 uses an underscore instead of the hyphen. ConfMiner flagged this commit, but unlike previous examples, since this is merely an issue of style, no reviewer specifically called this out. In fact, past interviews we have conducted have shown that reviewers hesitate from calling out such nits to maintain professional courtesy. We believe that automating such flagging of format issues can greatly improve readability and hygiene for configuration, while directing potential ire from developers at a bot rather than a human reviewer.

F. Real-world misconfigurations

To see how often such pattern-based misconfigurations occur in the wild, we manually examined the CTest dataset [20] which contains 64 real-world configuration-induced failures collected from 5 open-source projects. We found that, of the 64 issues, 51 were due to misconfiguration while the remaining 13 were due to bugs in code that parsed the configurations.

We studied the 51 misconfigurations in detail, and found 27 misconfigurations could potentially be caught using pattern mining. Of these, 18 misconfigurations were due to incorrect string formats in the value specified. For example, in `HDFS-7359`, the configuration parameter has to be parsed as an http address: any other string would cause a failure. The remaining 9 were due to numerical values lying outside a permissible range. For example, in `HBASE-13320`, the configuration parameter should either be a float value less than 1.0 or an integer greater than 1. Regular expressions can capture such specification.

However, in the other 24 cases, we found that regular expressions would not capture the specification for the configuration. For example, in issue `ZOOKEEPER-2264`, the user has to specify two configuration parameters or neither: specifying only one of the two caused an error. In issue `HADOOP-6566`, a configuration parameter has to be set to a directory path, and not a file path. Regular expressions cannot tell a file apart from a directory. Hence, of all misconfigurations, 53% (27 of 51) were amenable to pattern mining. With enough training examples, ConfMiner can learn such patterns.

V. Threats to Validity

As mentioned in Section II-C, ConfMiner is a best-effort system built on an inherent assumption that new configuration values will be similar to previous ones. Hence if a developer knowingly changes the value format, ConfMiner will not find a match and hence will generate a false-positive. However, this is unavoidable unless the developer provides hints to ConfMiner before-hand that the pattern is about to change. Manual input of this nature, while useful, is error-prone and does not scale to large services that uses millions of configuration parameters. Hence our effort has been to drive false-positives down as much as possible by fine-tuning the string profiling algorithm.

Also, since we depend upon commit histories and our difference module that performs a syntactic analysis of configuration files, for our implementation to be effective, configuration should be stored in well-formed files that are easy to parse, and well separated from code. If specification of configuration is inter-twined with code, or if they use non-standard formats, it becomes difficult to fine-tune our difference module to do the required syntactic analysis. While we do see examples of such scenarios, in most cases, we observe that developers maintain good hygiene and keep configuration and code files separate.

Our approach only considers the syntactic format of configuration values and ignores the semantics altogether. For example, even when the format of a timeout parameter is correct, the value may be incorrect in practice due to being very large or very small. In practice, it is not feasible to build a general and completely automated configuration mining tool that takes semantics into account.

VI. Related Work

A. Configuration Management

Previous work has used configuration files to learn “correct" data types [26] and flag misconfigurations when they occur. Configuration SpellCheck [27], [28] uses program analysis to detect configuration data types as well. We believe that detecting patterns based on data-types is very useful, but cannot capture the nuanced, fine-grained and varied patterns in configuration that are prevalent in today’s large-scale services, as shown in Table II. Further, apart from the basic data-types, Configuration SpellCheck requires the user to manually enter regular expression specifications for each configuration pattern. We believe that the number of configuration parameters in modern systems make manual authoring of specifications difficult, or impossible. Several other tools exist to check and validate a configuration file against a given specification [29], [30]. However, most specification is high-level and has to be manually entered by developers.

Recent work has focused on multiple data-driven and program analysis-based techniques to detect various different kinds of misconfiguration. REX [15] and Encore [31] use association rule mining to detect configurations that are correlated and flag misconfigurations based on the learned rules. PCheck [31], by performing static analysis on code, generates fast configuration checkers that emulate the code that uses the configuration. Code [32] analyzes event logs to detect anomalous event sequences and flag potential errors in configuration. Though not related to configuration, Getafix [33] uses pattern mining in code to detect missing null-reference checks in code. All these techniques are complementary to the program synthesis-based approach we take.

Several tools([34], [35], [36]) address how large services run by Facebook, Microsoft and Akamai have dealt with the problem of configuration management. These tools help engineers manage configuration across large deployments that span several geographies. A number of commercially available third-party tools also target configuration management [37], [38]. Facebook’s holistic configuration [34] specifically illustrates the effort required to detect misconfigurations early, by using automated canary testing for changed configurations, and using user-defined invariants to drive configuration changes.

We believe that techniques such as ours can work well in tandem with such configuration management systems to check for correctness before deploying configuration widely.

B. Profiling and Program Synthesis

Previous work on data profiling has focused more on statistical profiling of numerical data [39], [40], [41], [42]. See [43] for a survey of techniques. Several works in the databases literature have considered mining specifications that relate the values of one attribute to values of another through functional dependencies [44], [45], [46]. While we focus more on string typed values of a single configuration parameter, one potential direction for future work is to extend the work to learn from both numerical and string data, possibly relating the configuration values of one parameter to another.

Program synthesis has recently found significant success in the data manipulation, cleaning, and transformation fields [7], [8], [47]. In these settings, the synthesis takes the form of programming-by-example where the user provides a few input-output examples. However, in the string profiling setting the user does not provide examples of each cluster in the profile—instead, the synthesizer predictively learns the profile. In this manner, string profiling is closer in nature to other predictive synthesis works in the domain for data extraction [12], [48].

C. Regular expression and Automata Learning.

The L^* algorithm [49] was the first technique that learned finite automata from examples. Many variants of L^* have been studied over the past few decades [50], [51], [52] extending it to other automata variants including non-deterministic finite automata [53], alternating automata [54], and symbolic automata [55]. However, unlike our technique, L^* and its variants depend on an active teacher, i.e., an oracle that can produce counter-examples to intermediate guesses made by the learning algorithm. There have also been recent works that learn regular expressions from natural language using both sequence-to-sequence models [56], [57] and program synthesis techniques [58], [59].

There are several key differences between the current work and previous techniques driven by the underlying setting and motivation. The setting of our problem requires a technique that can learn multiple simple regular expressions that together match the examples as opposed to a single complex one, while ignoring noise in the provided examples. FlashProfile [13] is able to produce multiple regular expressions. However, as depicted in Section III, our technique produces higher quality profiles more efficiently as compared to FlashProfile. L^* and other language theoretic algorithms optimize either the size of the output automata or regular expression or minimality under language inclusion, resulting in complex and over-fitted regular expressions, making them unsuitable for our purposes.

VII. Conclusion

We have described a string profiling algorithm that learns various patterns in configuration used by large services. We have realized this through a tool called ConfMiner which is deployed on four repositories that maintain configuration for a large enterprise service. Using two sets of data that are available through version control systems – file-based and history-based – we show that our techniques learn a large number of varied patterns in configuration. These patterns capture various kinds of semantics thereby making the case for a generic algorithm that works across multiple domains. Finally, we also show that using these patterns, we can capture various kinds of misconfiguration at commit-time.

References

[1] Microsoft, "Microsoft 365." https://www.microsoft.com/microsoft-365. [Online; accessed 24-August-2021].

[2] SalesForce, "Acccess issue: May and june 2019." https://status.salesforce.com/incidents/3822. [Online; accessed 28-August-2020].

[3] Google, "Google cloud networking incident 19009." https://status.cloud.google.com/incident/cloud-networking/19009. [Online; accessed 28-August-2020].

[4] Sophos, "The state of cloud security 2020." https://secure2.sophos.com/en-us/medialibrary/pdfs/whitepaper/sophos-the-state-of-cloud-security-2020-wp.pdf. [Online; accessed 28-August-2020].

[5] O. Moolchandani, "Cloud waterhole - a novel cloud attack observed on twilio." https://www.linkedin.com/pulse/cloud-waterhole-novel-attack-observed-twilio-om-moolchandani/. [Online; accessed 28-August-2020].

[6] M. S. R. Center, "Access misconfiguration for customer support database." https://msrc-blog.microsoft.com/2020/01/22/access-misconfiguration-for-customer-support-database/. [Online; accessed 28-August-2020].

[7] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011* (T. Ball and M. Sagiv, eds.), pp. 317–330, ACM, 2011.

[8] V. Le and S. Gulwani, "Flashextract: a framework for data extraction by examples," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014* (M. F. P. O'Boyle and K. Pingali, eds.), pp. 542–553, ACM, 2014.

[9] R. Singh, "Blinkfill: Semi-supervised programming by example for syntactic string transformations," *Proc. VLDB Endow.*, vol. 9, no. 10, pp. 816–827, 2016.

[10] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig, "Trinity: An extensible synthesis framework for data science," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 1914–1917, 2019.

[11] A. S. Iyer, M. Jonnalagedda, S. Parthasarathy, A. Radhakrishna, and S. K. Rajamani, "Synthesis and machine learning for heterogeneous extraction," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019* (K. S. McKinley and K. Fisher, eds.), pp. 301–315, ACM, 2019.

[12] M. Raza and S. Gulwani, "Automated data extraction using predictive program synthesis," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA* (S. P. Singh and S. Markovitch, eds.), pp. 882–890, AAAI Press, 2017.

[13] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. D. Millstein, "Flashprofile: a framework for synthesizing data profiles," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 150:1–150:28, 2018.

[14] Google, "Google app engine app.yaml reference." https://cloud.google.com/appengine/docs/standard/python/config/appref. [Online; accessed 28-August-2020].

[15] S. Mehta, R. Bhagwan, R. Kumar, C. Bansal, C. Maddila, B. Ashok, S. Asthana, C. Bird, and A. Kumar, "Rex: Preventing bugs and misconfiguration in large services using correlated change analysis," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, (Santa Clara, CA), pp. 435–448, USENIX Association, Feb. 2020.

[16] Visual Studio Code, "Intellisense in visual studio code." https://code.visualstudio.com/docs/editor/intellisense. [Online; accessed 24-April-2019].

[17] "Content assist in eclipse." https://www.eclipse.org/pdt/help/html/code_assist.htm. [Online; accessed 28-August-2020].

[18] "Stylecop analyzers for the .net compiler platform." https://github.com/DotNetAnalyzers/StyleCopAnalyzers. [Online; accessed 28-August-2020].

[19] Microsoft, "Microsoft program synthesis using examples (prose) sdk." https://www.microsoft.com/en-us/research/group/prose. [Online; accessed 28-August-2020].

[20] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing configuration changes in context to prevent production failures," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 735–751, USENIX Association, Nov. 2020.

[21] The Git Version Control System. https://git-scm.com/.

[22] GitHub Inc. https://github.com. [Online; accessed 24-April-2019].

[23] Microsoft Azure DevOps. https://azure.microsoft.com/en-in/services/devops/. [Online; accessed 24-April-2019].

[24] Microsoft, "Generating diffgrams of xmlfiles." https://www.nuget.org/packages/XMLDiffPatch/. [Online; accessed 24-April-2019].

[25] Microsoft Azure Cloud Services. https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-choose-me. [Online; accessed 24-April-2019].

[26] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *Computer Aided Verification* (S. Chaudhuri and A. Farzan, eds.), (Cham), pp. 80–87, Springer International Publishing, 2016.

[27] A. Rabkin, "Using program analysis to reduce misconfiguration in open source systems software," tech. rep., Electrical Engineering and Computer Sciences, University of California at Berkeley, 2012.

[28] A. Rabkin, "The conf_spellchecker tool." https://github.com/roterdam/jchord/tree/master/conf_spellchecker. [Online; accessed 28-August-2020].

[29] "Validating xml files using xsd in c#." https://www.c-sharpcorner.com/article/how-to-validate-xml-using-xsd-in-c-sharp/. [Online; accessed 28-August-2020].

[30] "Configcop: A swift command line application that verifies .xcconfig files against a template.." https://github.com/fivegoodfriends/ConfigCop. [Online; accessed 28-August-2020].

[31] 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 (OSDI 16)*, (Savannah, GA), pp. 619–634, USENIX Association, 2016.

[32] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (USA), p. 28, USENIX Association, 2011.

[33] A. Scott, J. Bader, and S. Chandra, "Getafix: Learning to fix bugs automatically," *CoRR*, vol. abs/1902.06111, 2019.

[34] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 328–343, ACM, 2015.

[35] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein, "Acms: The akamai configuration management system," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, (Berkeley, CA, USA), pp. 245–258, USENIX Association, 2005.

[36] 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*, EuroSys '15, (New York, NY, USA), pp. 19:1–19:16, ACM, 2015.

[37] "The puppet configuration management tool." https://puppet.com/. [Online; accessed 28-August-2020].

[38] "Ansible for it automation." https://www.ansible.com/. [Online; accessed 28-August-2020].

[39] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Found. Trends Databases*, vol. 4, p. 1–294, Jan. 2012.

[40] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes, "Sampling-based estimation of the number of distinct values of an attribute," in *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, (San Francisco, CA, USA), p. 311–322, Morgan Kaufmann Publishers Inc., 1995.

[41] Y. Ioannidis, "The history of histograms (abridged)," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, p. 19–30, VLDB Endowment, 2003.

[42] P. Karras and N. Mamoulis, "The haar+ tree: A refined synopsis data structure," in *2007 IEEE 23rd International Conference on Data Engineering*, pp. 436–445, 2007.

[43] Z. Abedjan, L. Golab, and F. Naumann, "Profiling relational data: A survey," *The VLDB Journal*, vol. 24, p. 557–581, Aug. 2015.

[44] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann, "Scalable discovery of unique column combinations," *Proc. VLDB Endow.*, vol. 7, p. 301–312, Dec. 2013.

[45] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann, "Functional dependency discovery: An experimental evaluation of seven algorithms," *Proc. VLDB Endow.*, vol. 8, p. 1082–1093, June 2015.

[46] Y. Zhang, Z. Guo, and T. Rekatsinas, "A statistical perspective on discovering functional dependencies in noisy data," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, (New York, NY, USA), p. 861–876, Association for Computing Machinery, 2020.

[47] R. Singh, "Blinkfill: Semi-supervised programming by example for syntactic string transformations," *Proc. VLDB Endow.*, vol. 9, p. 816–827, June 2016.

[48] M. Raza and S. Gulwani, "Web data extraction using hybrid program synthesis: A combination of top-down and bottom-up inference," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020* (D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, eds.), pp. 1967–1978, ACM, 2020.

[49] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, p. 87–106, Nov. 1987.

[50] R. Parekh and V. Honavar, "An incremental interactive algorithm for regular grammar inference," in *International Colloquium on Grammatical Inference*, pp. 238–249, Springer, 1996.

[51] R. Parekh and V. Honavar, "Learning dfa from simple examples," *Machine Learning*, vol. 44, no. 1, pp. 9–35, 2001.

[52] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, (New York, NY, USA), p. 411–420, Association for Computing Machinery, 1989.

[53] F. Denis, A. Lemay, and A. Terlutte, "Learning regular languages using non deterministic finite automata," in *ICGI*, 2000.

[54] D. Angluin, S. Eisenstat, and D. Fisman, "Learning regular languages via alternating automata.," in *IJCAI*, pp. 3308–3314, 2015.

[55] S. Drews and L. D'Antoni, "Learning symbolic automata," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I* (A. Legay and T. Margaria, eds.), vol. 10205 of *Lecture Notes in Computer Science*, pp. 173–189, 2017.

[56] N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman, and R. Barzilay, "Neural generation of regular expressions from natural language with minimal domain knowledge," 08 2016.

[57] Z. Zhong, J. Guo, W. Yang, J. Peng, T. Xie, J.-G. Lou, T. Liu, and D. Zhang, "Semregex: A semantics-based approach for generating regular expressions from natural language specifications," in *EMNLP*, 2018.

[58] X. Ye, Q. Chen, X. Wang, I. Dillig, and G. Durrett, "Sketch-driven regular expression generation from natural language and examples," *Trans. Assoc. Comput. Linguistics*, vol. 8, pp. 679–694, 2020.

[59] M. Lee, S. So, and H. Oh, "Synthesizing regular expressions from examples for introductory automata assignments," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, (New York, NY, USA), p. 70–80, Association for Computing Machinery, 2016.