A Novel Method to Prevent Misconfigurations of Industrial Automation and Control Systems

Yu Zhang, Yani Ge, Peiran Yu, Jianzhong Zhang, Yongzheng Zhang, and Thar Baker

***Abstract*—Configuration errors are among the dominant causes of system faults for the industrial automation and control systems (IACS). It is difficult to detect and correct such errors of IACS as there are various kinds of systems and devices with miscellaneous configuration specifications. In this paper, we first propose a streaming algorithm to keep all the configuration changes in the limited memory space. And, when making a new configuration change, another novel streaming algorithm is proposed to search and return all the similar historical changes which can be used to validate this new one. So far, we are the first to model the configuration changes of IACS as a data stream and apply the streaming similarity search in correcting configuration errors while overcoming the inherent unbounded-memory bottleneck. The theoretical correctness and complexity analyses are presented. Experiments with real and synthetic datasets confirm the theoretical analyses and demonstrate the effectiveness of the proposed method in preventing misconfigurations of IACS.**

***Index Terms*—industrial automation and control systems, configuration management, similarity search, data stream.**

I. Introduction

INDUSTRIAL automation and control systems (IACS) are typically used in industrial electric, water, oil, transportation, chemical, pharmaceutical, discrete manufacturing and so on. These systems work mutually dependently based on a wide variety of Industrial Internet of Things (IIoT) devices for sensing and actuation. Nowdays, IACS are facing more and more cybersecurity issues [1], [2], [3], [4]. In order to assure that the systems can run exactly as expected, it is important to guarantee the correctness of the configurations of IACS as well as the massive IIoT devices.

However, the configuration management in IACS is much more complicated as there are a variety of systems and IIoT devices with different configuration specifications. In all the misconfiguration errors, there is a significant percentage of illegal configuration parameters and typos. As a result, it is useful to propose an effective misconfiguration avoidance method to reduce all these unnecessary mistakes.

Manuscript received April 18, 2020; revised July 10, 2020; accepted August 7, 2020. This work was supported by the National Key R&D Program of China under Grant 2018YFB0804702, and by the Technology R&D Program of Tianjin under Grant 18ZXZNGX00200 and Grant 18ZXZNGX00140. Paper no. TII-20-1986. (Corresponding author: Yu Zhang.)

Y. Zhang, Y. Ge, P. Yu and J. Zhang are with the College of Cyber Science, College of Computer Science, Tianjin Key Laboratory of Network and Data Security Technology, Nankai University, Tianjin 300350, China. (e-mail:zhangyu1981@nankai.edu.cn; yanige@outlook.com; peiranyu@outlook.com; zhangjz@nankai.edu.cn)

Y.Z. Zhang is with the Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. (e-mail:zhangyongzheng@iie.ac.cn)

T. Baker is with the Liverpool John Moores University, United Kingdom. (e-mail:t.baker@ljmu.ac.uk)

The intuition of the proposed method is that all the configuration files can be transformed into the format of key-value pairs, where the key corresponds to a single configuration directive consistently across different system instances. And then for a specific configuration directive, its correct key-value pairs will have high similarity obviously. For example, in the Apache configuration file $httpd.conf$, in most cases the usual value of the configuration directive $LoadModule\ my\_mod$ is $my\_mod\_file.so$, and the usual value of the configuration directive $Listen$ is 8080, while in the MySQL configuration file $my.cnf$, in most cases the usual value of the configuration directive $mysqld\ datadir$ is $/var/lib/mysql$. However, when encountering the misconfigurations, there would be miscellaneous incorrect key-value pairs, therefore the similarity will be very low apparently. As we can see, the key point of the proposed misconfiguration avoidance method is the string similarity search over data stream.

A. Related Work

For traditional information systems, there have been a lot of related works in configuration management. Generally speaking, related works can be classified into three categories: helping the users to fix configuration problems, validating the actions when changing configurations, automating the configuration management. Some researches, such as [5], [6], [7], use computers to troubleshoot the configuration problems. The basic idea of Chronus [5] is searching for the historical working state. Chronus logs every change to the disk. When diagnosing a configuration problem, Chronus loads and runs historical system snapshots, and tests whether the historical system snapshots work correctly. Autobash in [6] helps a user to locate existing solutions to configuration problems. Autobash maintains a database of user actions to known configuration problems and uses a trial-and-fail approach to test candidate solutions. PeerPressure [7] uses Bayesian statistical analysis to find the misconfiguration root causes. PeerPressure keeps a database storing the system state of many computers. When diagnosing the configuration problem, PeerPressure first finds suspect registry entries that may cause the problem. Then PeerPressure retrieves the same set of entries from other computers. Finally, PeerPressure uses the Bayesian statistical estimations to calculate the probabilities for all these entries. Some other researches, as in [8], [9], provide frameworks to validate configuration changes before they are put in effect on production systems. Nagaraja et al. [8] develop a validation framework which can detect operator mistakes before deployment by comparing against the comparator functions provided by users. Oliveria et al. [9] validate

database system administrations. They introduce a new model-based validation technique in its validation framework. Other researches [10], [11] use computers to accomplish the error-prone configuration management tasks. They propose some high-level language directives or templates that define how the configuration files should be generated. SmartFrog [10] uses a declarative language to describe software components and configuration parameters. Zheng et al. [11] leverage custom-specified templates to automatically generate the correct configuration for a system.

As a well-known problem in data mining, the purpose of string similarity search is to find all strings within a given edit distance from the query string in a set of strings [12], [13], [14], [15], [16], [17]. However, most related researches focus on building the index of a fixed size set of strings to improve the performance of query [14], [15], [16], [17]. Only a few works have been done on data stream, and most of them focus on time series [18], [19], and most use the sliding window model [12] which is apparently different from the landmark model used in this paper.

B. Identified Challenges

String similarity search over the landmark data stream model brings the following challenges:

- The first is how the arriving strings should be stored. At present, in most data stream problems, the Bloom Filter(BF) structure is used to store data [20]. However, only using BF will limit the edit distance threshold and waste a lot of available memory space in the computer.
- The second is how the deleted strings should be recovered. In the data stream environment, it is inevitable to remove some arrived strings to save more memory space for newly arriving ones. However, when a new query arrives, we need to find a method to recover the historically deleted strings.

C. Contributions

The above challenges are solved in this paper and the contributions of this paper can be summarized as:

- Two clustering-based string compression algorithms are proposed to solve the problem of continuous stream arrival vs. the insufficient memory space.
- A historical string recovery algorithm based on Bloom Filter is proposed so that the deleted strings can be recovered without omission when a query arrives.
- A brand new misconfiguration avoidance method based on streaming similarity search is proposed to aid diagnosing configuration changes of IACS.

D. Organization

The remainder of this paper is organized as follows. In Section II, we introduce some preliminaries, including some definitions and theorems. Then, we illustrate the details of the proposed streaming similarity search method as well as the misconfiguration avoidance method in Section III. In Section IV, we present the experiment results and evaluation. Finally, the conclusion is given in Section V.

II. Preliminaries

A. Problem Definition

Given N configuration directives of all systems and IIoT devices, and let D be the set of such directives denoted as $D = \{d_1, d_2, d_3, \cdots, d_{N-1}, d_N\}$, where each d_i $(1 \le i \le N)$ is a configuration directive name. We model the configuration changes as a data stream $S = (s_1, s_2, \cdots, s_n, \cdots)$, where $s_n = d||v_{d,n}$ is a string concatenation of d and $v_{d,n}$. Usually, $d \in D$ is a configuration directive name. However, there might be some typos or spelling errors. In this case, although $d \notin D$, d is very similar to the correctly typed configuration directive in D. n is the index of the change, $v_{d,n}$ is the value set for d at the n-th change. The string $d||v_{d,n}$ means the n-th change is made to the configuration directive d with value $v_{d,n}$. The basic idea of the proposed misconfiguration avoidance method is described as follows. Assume that from the very beginning there have been n changes in the configurations of all systems and IIoT devices. The proposed method keeps all n changes (i.e., $S = (s_1, s_2, \cdots, s_n)$) in memory. When comes a new change s_{n+1}, the misconfiguration avoidance method first searches the configuration data stream S and then returns all the similar changes to s_{n+1}. Generally speaking, in a well-performed system, the vast majority of changes to the configurations should be correct, therefore the returned similar changes will be of great reference value for the new change s_{n+1}, e.g., they can be used to check whether there exist typos in s_{n+1} or not. In other words, if there are no similar changes returned, then the new change s_{n+1} demands careful consideration, i.e., there might be some errors in s_{n+1}.

Let Σ denote a finite alphabet and s is a string made up of a sequence of letters from Σ. In this paper, we use edit distance to quantify the similarity between two strings. Edit distance $ED(s_1, s_2)$ is the minimum number of edit operations (insertion, deletion and substitution) required to change from s_1 to s_2. By using dynamic programming, the edit distance can be computed in $O(n^2)$ time with $O(n)$ space [21].

Definition 1 (Similarity Search). *Given a set of strings S, a query string q, a threshold of edit distance τ, Similarity Search is to find all s from S, the edit distance between s and q is less than or equal to τ.*

Definition 2 (String Stream). *A String Stream is a specific type of data stream of which the dynamic and infinite arrivals are in the form of strings.*

In this paper, we adopt a landmark window model for the string stream which starts at a fixed time and never ends, which means the beginning of the string stream is fixed but the end grows.

Definition 3 (Similarity Search over a String Stream). *Given a string stream S starting at a certain time, when comes a query string q and gives a threshold of edit distance $\tau(\tau \ge 1)$, the goal of similarity search over a string stream is to find all $s \in S$ similar to q till now, i.e., $ED(q, s) \le \tau$.*

B. Theorems

Given strings q, s, c consisting of letters in Σ and edit distance thresholds τ, let $S = \{s|ED(s, c) \le r'\}$ denote the strings clustered by c with radius r'.

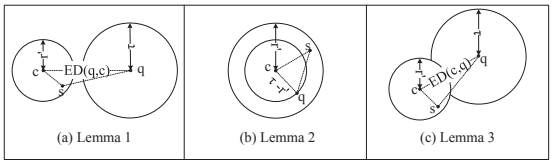

Fig. 1. Illustration of theorems. (a) illustrates the situation that satisfies the condition $ED(q,c) > \tau + r'$, $ED(c,s) \leq r'$; (b) illustrates the situation that satisfies the condition $ED(q,c) \leq \tau - r'$, $0 \leq ED(c,s) \leq r'$; (c) illustrates the situation that meets the condition $|ED(c,s) - ED(c,q)| < ED(q,s)$, $|ED(c,s) - ED(c,q)| > \tau$.

Theorem 1: Given an edit distance function ED and strings s_1, s_2, s_3, there is $|ED(s_1,s_2) - ED(s_2,s_3)| \leq ED(s_1,s_3) \leq ED(s_1,s_2) + ED(s_2,s_3)$.

Lemma 1: When $ED(q,c) > r' + \tau$, it must be true that $ED(q,s) > \tau$, $s \in S$.

Proof: As Fig.1(a) shows, $ED(q,c) > r' + \tau$, $ED(c,s) \leq r'$. Therefore, we have $|ED(q,c) - ED(c,s)| > \tau$. Since $|ED(q,c) - ED(c,s)| \leq ED(q,s) \leq ED(q,c) + ED(c,s)$, we have $ED(q,s) > \tau$. ■

Lemma 2: When $ED(q,c) \leq \tau - r'$, it must be true that $ED(q,s) \leq \tau$, $s \in S$.

Proof: As Fig.1(b) shows, $ED(q,c) \leq \tau - r'$, $0 \leq ED(c,s) \leq r'$. Therefore, we have $ED(q,c) + ED(c,s) \leq \tau$. Since $|ED(q,c) - ED(c,s)| \leq ED(q,s) \leq ED(q,c) + ED(c,s)$, we have $ED(q,s) \leq \tau$. ■

Lemma 3: When $|ED(c,s) - ED(c,q)| > \tau$, it must be true that $ED(q,s) > \tau$, $s \in S$.

Proof: As Fig.1(c) shows, $|ED(c,s) - ED(c,q)| < ED(q,s)$, $|ED(c,s) - ED(c,q)| > \tau$. Therefore, we have $ED(q,s) > \tau$. ■

C. Bloom Filter

Bloom Filter (BF) [20], [22] is a data structure based on hash, which stores data in a fixed number of bits. A BF contains m' bits, represented by $BF[0], BF[1]...BF[m'-1]$, each bit is initialized to 0. It is used to represent a set S' that includes n elements. There are k independent hash functions $h_1, h_2...h_k$, the value of each hash function ranges in $[0, m'-1]$. Assume that these hash functions independently and uniformly map each element to a random number in the entire range. For $a \in S'$, set $BF[h_i(a)]$ to 1 for $1 \leq i \leq k$. For b, check each $BF[h_i(b)](1 \leq i \leq k)$ whether it is 1 or not. If every $BF[h_i(b)](1 \leq i \leq k)$ is 1, then b may belong to S'. As the bit can be set to 1 by any other strings, so we cannot ensure it is certainly set by b. But finding one bit $BF[h_i(b)] = 0$ implies certainly $b \notin S'$, since if b belongs to the set, every bit $BF[h_i(b)](1 \leq i \leq k)$ will be set to 1. This explains why Bloom Filter scheme has false positive. Let n be the number of strings in set S'. The hash function maps string randomly and uniformly in $[0, m'-1]$. Let p denote the probability that a random bit of BF is 0, the false positive rate is denoted as:

$$f = (1-p)^k \approx (1 - e^{-kn/m'})^k \quad (1)$$

We can reduce the value of f by choosing an appropriate m' when the number of set n is known.

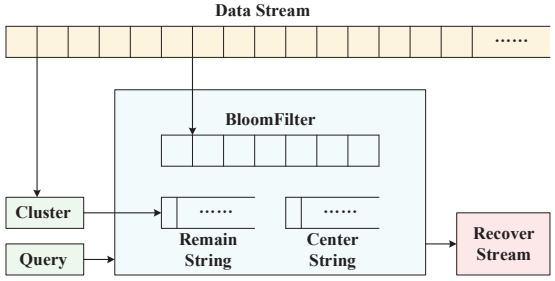

Fig. 2. The overall process of streaming similarity search algorithm. First, all the strings coming through will be recorded in a BF and stored in memory. Then, when the number of strings reaches an upper limit, do clustering and only the centroids and outliers will be remained. Finally, do an approximate query on the string stream.

III. Similarity Search over a String Stream

A. Basic Idea

Current string similarity search algorithms are suitable for processing static strings. When comes a dynamic string stream, the major problem becomes how to store the infinite strings as well as how to process the query efficiently. In this paper, we propose an approach to overcome the inherent unbounded-memory bottleneck. The basic idea is to use a BF maintained in memory to record all the strings coming through. At the very beginning, all the arrived strings are stored in memory as many as possible. When the number of preserved strings reaches an upper limit, a batch approach is adopted to delete some strings by utilizing the clustering-based string compression algorithms. As a result, only the centroids and outliers (i.e., the uncompressed strings) will be kept in memory. And an approximate query on the string stream will be performed based on the BF as well as the preserved strings (i.e., the centroids and outliers). The overall process is illustrated in Fig.2. In this way, the dynamic string stream is converted to a static string set.

Given a query string q, a threshold of edit distance τ, the basic idea of querying similar strings over the data stream (Algorithm 1) is described as follows.

1) If $\tau \leq r'$, find every string similar to q till now by the BF-based string recovery algorithm $recoverBF(q, 0, \tau, BF)$ (Algorithm 5) and add them to the result set, where BF refers to the Bloom Filter, r' is the radius of clustering, determined by the processing power of the computing platform. Otherwise ($\tau > r'$), turn to step 2.

2) Use existing string similarity search algorithms (e.g., Pass-Join [21]) to find every $outlier$ preserved in memory satisfying the condition $ED(q, outlier) \leq \tau$, and add them to the result set. Go to step 3.

3) For each $centroid$ preserved in memory, if $ED(q, centroid) > r' + \tau$, meaning strings compressed by this $centroid$ with radius r' will not overlap with the strings meeting the query condition according to *Lemma 1*. Therefore, we do not need to recover strings compressed by this $centroid$. Go to step 4.

4) If $ED(q, centroid) \leq \tau - r'$, meaning each string compressed by this $centroid$ is similar to q according to *Lemma 2*, we need to recover the strings using

Algorithm 1: Query(q, τ)

Input: The query string q, edit distance threshold τ
Output: The result string set R

```
if τ ≤ r' then
    //BF denotes the Bloom Filter
    R ← recoverBF(q, 0, τ, BF);
else
    for each outlier in Outliers do
        if ED(q, outlier) ≤ τ then
            Add outlier into R;
    for each centroid in Centroids do
        if ED(q, centroid) > τ + r' then
            //Lemma1
            continue;
        else if ED(q, centroid) ≤ τ − r' then
            //Lemma2
            R ← R ∪ recoverBF(centroid, 0, r', BF);
        else
            l ← Max(0, ED(q, centroid) − τ);
            for each str returned by
              recoverBF(centroid, l, r', BF) do
                if ED(q, str) ≤ τ then
                    Add str into R;
```

$recoverBF(centroid,\ 0,\ r',\ BF)$, but need not to calculate the edit distance. Go to step 5.

5) Other than that, it's impossible to judge the relationship between strings compressed by this $centroid$ and the query string q. Firstly, we need to recover all strings compressed by this $centroid$ using $recoverBF(centroid, l, r', BF)$, where $l = ED(q, centroid) - \tau$ if $ED(q, centroid) - \tau \geq 0$, otherwise $l = 0$. And then calculate the edit distances between recovered strings and the query string q. Add the similar strings to the result set.

The BF-based string recovery algorithm $recoverBF$ exhibits an exponential time complexity (see subsection III-C). Considering the processing power of current computers, the radius r' cannot be too large. In our experiment environment, $r' \leq 5$. To increase the query scope, we need to expand the query edit distance threshold. As a result, two clustering-based compression algorithms are proposed to keep historical strings in memory as many as possible. When exceeding a certain number, it does clustering, only the centroids and outliers are remained, and an approximate query on the string stream is performed based on these reserved strings.

B. Clustering-based String Compression Algorithm

Given a string set $S = \{s_1, s_2 \cdots s_n\}$, the outputs of the clustering-based compression algorithm are the centroid set C and the outlier set O, where $C = \{(c_1, r_1), (c_2, r_2) \cdots (c_t, r_t)\}$, $c_i \in S$ is a centroid, r_i is its respective clustering radius ($1 \leq i \leq t, t \leq n$), $O = \{o_1, o_2 \cdots o_l\}$, $o_j \in S$ is an outlier ($1 \leq j \leq l, l \leq n$). In general, t and l are far less than n.

In this paper, two clustering-based compression algorithms are proposed: one is the set cover based clustering algorithm, and the other is the VP-Tree based clustering algorithm. The former has better clustering effect, and the latter has a lower time complexity.

1) Set Cover Based Clustering Algorithm: The minimum set cover problem is described as below: *Given a universal set U, and the finite subset of U: $S_1, S_2 \ldots S_w$, the problem is to find out the minimum number of subsets that the union of them is U, which means to find the smallest $I \subseteq \{1, 2 \ldots w\}$ makes $\cup_{i \in I} S_i = U$.*

The minimum set cover problem has been proved to be an *NP-hard* problem [23], [24]. An approximate solution is obtained with the greedy algorithm [25]. The basic idea is choosing the subset that includes the maximum uncovered elements every time till all elements are covered. This algorithm has been proved to be a polynomial time complexity approximate algorithm of the set cover problem. The minimum coverage found by this greedy algorithm may be $H(s)$ times as large as the real minimum coverage, where $H(s) = \sum_{k-1}^{s} \frac{1}{k} \leq ln(s+1)$, s is the size of the set to be covered.

Algorithm 2: SetCover Clustering

Input: String set S, clustering radius r'
Output: Centroid set C, outlier set O

```
for i = 0 → S.size do
    Add S[i] to distSet[i] ;
    for each S[j] ∈ S and ED(S[i], S[j]) ≤ r', i ≠ j do
        Add each S[j] to distSet[i];
//see SetCover in Algorithm 3
covSet = SetCover(distSet);
for j = 0 → covSet.size do
    k ← covSet[j].maxIndex;
    subset ← covSet[j].maxSet;
    if subset.size = 1 then
        Add the only string in subset into O;
    else
        Add (S[k], r') into C;
```

The SetCover clustering algorithm is depicted in Algorithm 2. At first, there are the string set $S = \{s_1, s_2 \ldots s_n\}$ and the clustering radius r'. Given the $distSet = \{distSet[1], distSet[2] \cdots distSet[n]\}$, where $distSet[i]$ is made up of s_i with all its similar strings, i.e., $distSet[i] = \{s_i\} \bigcup \{s_j | ED(s_j, s_i) \leq r', i \neq j\} (1 \leq i \leq n)$, the SetCover algorithm (depicted in Algorithm 3) takes $distSet$ as input, and produces $covSet$ which is composed of the resulting subsets. For each subset in $covSet$, if this subset has only one string, then add this string to the set of outliers, i.e., the set of unclustered strings O. If not, which means there are strings that can be clustered by s_k which is a centroid, then add s_k to the set of centroids C, where k is the corresponding index of this subset in $distSet$.

For example, $S = \{a, aaa, b, ccc, ddddd\}$, $r' = 3$, $distSet = \{distSet_1 : \{a, aaa, b, ccc\}, distSet_2 : \{aaa, a, b, ccc\}, distSet_3 : \{b, a, aaa, ccc\}, distSet_4 : \{ccc, a, aaa, b\}, distSet_5 :$

$\{ddddd\}\}$. After calculation, we can get $C = \{(a,3)\}$, $O = \{ddddd\}$.

Algorithm 3: SetCover

Input: $distSet$
Output: $covSet$
while *non-empty subsets left in* $distSet$ **do**
  $maxNum = 0$;
  for $i = 0 \rightarrow distSet.size$ **do**
    if $distSet[i].size > maxNum$ **then**
      $maxNum \leftarrow distSet[i].size$;
      $maxSet \leftarrow distSet[i]$;
      $maxIndex \leftarrow i$;
  $distSet[maxIndex] \leftarrow \emptyset$;
  for $j = 0 \rightarrow distSet.size$ **do**
    delete all s in $distSet[j]$, $s \in maxSet$;
  Add ($maxSet$, $maxIndex$) into $covSet$;

The basic idea of SetCover depicted in Algorithm 3 is as follows: choose the subset in $distSet$ that includes the most strings as $maxSet$; empty this chosen subset in $distSet$; for each $s \in maxSet$, delete s from each subset in $distSet$; repeat this process till all the subsets in $distSet$ are empty.

For example, $distSet = \{distSet_1 : \{a, aaa, b, ccc\}, distSet_2 : \{aaa, a, b, ccc\}, distSet_3 : \{b, a, aaa, ccc\}, distSet_4 : \{ccc, a, aaa, b\}, distSet_5 : \{ddddd\}\}$. At first, choose $maxSet = distSet_1 : \{a, aaa, b, ccc\}$, after deleting all elements within $maxSet$ from $distSet$, we get $distSet = \{distSet_1 : \emptyset, distSet_2 : \emptyset, distSet_3 : \emptyset, distSet_4 : \emptyset, distSet_5 : \{ddddd\}\}$, and $covSet = \{(\{a, aaa, b, ccc\}, 1)\}$. And then, choose $maxSet = distSet_5 : \{ddddd\}$, after deletion, $distSet = \{distSet_1 : \emptyset, distSet_2 : \emptyset, distSet_3 : \emptyset, distSet_4 : \emptyset, distSet_5 : \emptyset\}$, $covSet = \{(\{a, aaa, b, ccc\}, 1), (\{ddddd\}, 5)\}$.

Complexity: Given a string set $S = \{s_1, s_2 \cdots s_n\}$, there are n subsets in $distSet$, and each subset is composed of at most n strings. Apparently, the time complexity to get the $distSet$ is $O(n^2 \times t_{ed})$, where t_{ed} is the time needed to compute the edit distance of a pair of strings. That is, it needs to compute the edit distances of at most n^2 string pairs, we can utilize Pass-Join Algorithm [21] to optimize the calculation process. After the compution of $distSet$, the time complexity to get the approximate minimum set cover according to the SetCover algorithm (Algorithm 3) is $O(n^2)$. As a result, the total time complexity of the SetCover clustering algorithm (Algorithm 2) is $O(n^2 \times t_{ed} + n^2)$. The original $distSet$ contains a total of n subsets, and each subset has a maximum of n strings, so the space complexity is $O(n^2)$.

2) VP-Tree Based Clustering Algorithm: The VP-Tree (Vantage Point Tree) structure [26] aims to solve the nearest neighbor problem in metric space. It is basically a binary tree. To construct it, every time we choose a point as a "vantage point", then calculate the edit distances between this "vantage point" and other points, and divide all other points to the left or right subtree according to the median of all the calculated edit distances. The edit distances between the "vantage point" and the points in the left subtree are no larger than the median, and the points in right are larger than the median.

Algorithm 4: VP-Tree Clustering

Input: String set S, clustering radius r'
Output: Centroid set C, outlier set O
$vpTree$=buildVpTree(S);
//preorder traversing the vpTree
while $vpTree$!= $NULL$ **do**
  Get the root vantage point vp in $vpTree$;
  if $vp.distance \leq r'$ **then**
    if $vp.leftsubtree$!= $NULL$ **then**
      Add ($vp.string, vp.distance$) into C;
    else
      Add $vp.string$ into O;
    Recursive search in $vp.rightsubtree$;
  else
    Add $vp.string$ into O;
    Recursive search in $vp.leftsubtree$;
    Recursive search in $vp.rightsubtree$;

When strings stored in memory reach a maximum capacity, they are built into a VP-Tree. And then we can utilize the VP-Tree clustering algorithm (Algorithm 4) to compress the strings and save more memory for new coming ones. The whole clustering process is depcited as follows. Firstly, we build the string set S into a VP-Tree. Each vantage point in the VP-Tree saves a median distance, and all nodes in its left subtree have edit distances less than or equal to the median, while all nodes in its right subtree have edit distances larger than the median. As long as the median distance is no larger than the clustering radius r' and the left subtree is not empty, then this vantage point will be regarded as a centroid, and the string corresponding to this vantage point as well as its median distance will be saved in the centroid set C (in this case, all the strings in its left subtree will be compressed), and recursive search will be performed in its right subtree only. Otherwise (median distance is larger than r'), the vantage point string will be saved in the outlier set O. Recursive search will be performed in its left subtree and right subtree respectively.

In fact, in order to compress more strings in the resulting outlier set O, we can rerun the VP-Tree clustering algorithm many times with the resulting outlier set O as the new input string set S. This process can be repeated till enough memory space is saved. The compression ratio rto is defined as the number of remaining strings (including centroids and outliers) divided by the total number of original strings:

$$rto = \frac{\mid O_k \mid + \sum_{i=1}^{k} \mid C_i \mid}{\mid S \mid}, i = 1, 2 \cdots k \tag{2}$$

Notes: where k is the number of times the VP-Tree clustering algorithm reruns (i.e., the number of clustering times), $\mid C_i \mid$ is the number of centroids in the i-th clustering, $\mid O_k \mid$ is the number of outliers in the k-th (last) clustering, $\mid S \mid$ is the number of strings in the original string set S.

TABLE I
PARAMETERS AND DESCRIPTION

Notation	Description
Σ	Alphabet
m	Number of alphabet letters
s	String
$\|s\|$	The length of the string s
q	Query string
r	Recovered string
τ	Edit distance threshold
$ED(q,r)$	Editing distance of strings q and r
x,y,z	Number of insertions, deletions and substitutions

Complexity: When building a VP-Tree, since all strings are saved in a list in the beginning, only one position operation is required for each random selection. And then we compute the edit distances between this chosen string and all other strings, and find the median by the binary search. Therefore, given a string set S composed of n strings, the time complexity of constructing a VP-Tree is $O(nlog(n)\times t_{ed})$, where t_{ed} is the time needed to compute the edit distance of a pair of strings. After building the VP-Tree, we need to traverse the entire tree. Specifically speaking, we need to check whether each vantage point in the VP-Tree can be regarded as a centroid or not, of which the time complexity is $O(n)$. Therefore, the total time complexity of the VP-Tree clustering algorithm (Algorithm 4) is about $O(nlog(n)\times t_{ed}+n)$. The space required is proportional to the number of strings, and the space complexity is $O(n)$.

C. Bloom Filter Based String Recovery Algorithm

This section mainly introduces the string recovery algorithm that satisfies the edit distance condition. The parameters required below are shown in the TABLE I.

Definition 4 (String Recovery). *Given a string q, a threshold of edit distance τ, and an alphabet Σ, the string recovery is to get the string set $R = \{r \in \sum^n |ED(q,r) \leq \tau\}$, $n \in [|q|-\tau, |q|+\tau]$.*

Algorithm 5: recoverBF(q,l,u,BF)

Input: The query string q, lower l, upper u, Bloom Filter BF
Output: A string set $R \supseteq \{r|r \in BF \;\&\&\; l \leq ED(q,r) \leq u\}$

```
1 R ← ∅ ;
2 for i = l; i ≤ u; i++ do
3     //x,y,z denote number of insertions, deletions
4     //and substitutions respectively
5     for each combination of x,y,z (x + y + z = i) do
6         R_i ← EditOperate(q, x, y, z, BF) ;
7         R ← R ∪ R_i ;
```

Because the deletions and substitutions are performed on the original string, the insertion operations should be performed at last. And in order to cover all cases and get the full string set, we need to do the substitutions at first. Therefore, the order of these three editing operations is: substitution, deletion and insertion.

The basic idea of the BF-based string recovery algorithm recoverBF (Algorithm 5) is described as follows: operate on the query string q in the order of substitution, deletion and insertion, and then check each candidate string in the BF. If the BF returns true, then add this string to the result string set R. If not, ignore this string. Note that in the final result set R, there might be some additional strings of which the edit distances with the query string q are less than the lower bound l. This is because that the $EditOperate(q,x,y,z,BF)$ might return a string set $R^{'}$ including some additional strings of which the edit distances are less than $x+y+z$. However, this will not affect the correctness of our proposed streaming similarity search algorithm (Algorithm 1), as we will verify the real edit distance of each string returned by recoverBF with the query string. Note that the definition of string recovery problem is the special case when $l=0$ and $u=\tau$ in recoverBF.

Algorithm 6: EditOperate(q,x,y,z,BF)

Input: The original string q, number of insertions/deletions/substitutions $x/y/z$, Bloom Filter BF
Output: A string set $R^{'} \supseteq \{r|r \in BF \;\&\&\; ED(q,r) = x(insertions)+y(deletions)+z(substitutions)\}$

```
1 R' = ∅;
2 for each string q' returned by Substitute(q, z) do
3     for each string q'' returned by Delete(q', y) do
4         for each string q''' returned by Insert(q'', x) do
5             if BF(q''') = True then
6                 Add q''' to R';
```

The detailed editing operation is depicted in EditOperate (Algorithm 6). The number of insertions, deletions and substitutions are x,y,z respectively. For substitution operation, z positions are selected from the original string q, and each position has m (number of alphabet letters in Σ) possibilities. Therefore, it is actually a combination problem and we can know the string q is replaced by z times to get $m^z C^z_{|q|}$ cases. Similar to substitution, deletion is also a combination problem. There are $C^y_{|q|}$ cases when we select y positions from the original string q to delete. The situation of insertion is relatively complicated. x insertions are equivalent to inserting different combinations of a string l of length x. The detailed operation is described as below: divide string l into different groups, for example, abc can be divided into four groups: $a\ b\ c, ab\ c, a\ bc, abc$. Obviously, there are $2^{|l|-1}$ groups. Suppose the number of elements in the i-th group is x_i, $(i \leq 2^{|l|-1})$, it is transformed to a combination problem of selecting x_i positions from the original string q. Finally, insert each group element at its corresponding position. The number of cases of x insertions to q is:

$$m^x \sum_{i=0}^{2^x-1} C^{x_i}_{|q|+1} \tag{3}$$

By the above analysis, in EditOperate, the number of strings generated from a string q after x insertions, y deletions

TABLE II
DATA SET DETAILS

Dataset	Cardinality	AvgLen	MaxLen	MinLen
Word	122823	8.722	29	5
Name	1957046	6.171	12	2
Pwd	1000000	7.529	39	1

and z substitutions is:

$$n_{edit}(q,x,y,z) = m^z C^z_{|q|} C^y_{|q|} m^x \sum_{i=0}^{2^x-1} C^{x_i}_{|q|-y+1} \quad (4)$$

When the edit distance is $k = x+y+z$, the number of all combinations of insertion, deletion, and substitution operations is $\sum_{i=0}^{k}(k-i+1) = \frac{(k+1)(k+2)}{2}$. Therefore, in recoverBF the maximum possible number of strings returned (i.e., in case all strings are in the Bloom Filter) is : $n_{recover}(q,l,u) = \sum_{k=l}^{u} \frac{(k+1)(k+2)}{2} \times n_{edit}(q,x,y,z)$.

Obviously, the total time complexity of the recoverBF algorithm (Algorithm 5) is $O(n_{recover}(q,l,u))$. Since recoverBF does not store every string it recovers, instead it searches directly in the BF structure, and returns the strings which are stored in BF only (see Algorithm 6), the memory space required by recoverBF is relatively small. It can be seen that the string recovery is an NP-hard problem, and the proposed method recoverBF is an exhaustive algorithm with high time complexity. In addition, since the completeness of the recovered strings must be guaranteed (no historical string omitted), no approximation algorithm can be used.

D. Misconfiguration Avoidance Method

Finally, the misconfiguration of IACS can be avoided in this way:

- First of all, employ the proposed string clustering algorithms to keep all the configuration changes of IACS in memory.
- When comes a new configuration change, we can check its correctness with the following steps: 1) Employ the proposed streaming similarity search algorithm to query its similar changes from all the historical configuration changes. 2) If returning some similar changes, we then can use them to evaluate the correctness of this new change. Generally speaking, if there are a lot of similar configuration changes returned, then this new change is believed to be correct with high probability. 3) Otherwise, if there are no similar changes returned, then this new change requires more attention and there might be some errors.

IV. EXPERIMENT

We first test our method with three public datasets as shown in Table II. The entire Word [21] and Name* datasets as well as part of the Pwd† (1 million strings) dataset are adopted in the experiments. So far as we know, we are the first to address the streaming similarity search problem, which means that there are no comparable related works. Therefore, we mainly evaluate the performance of the proposed method, including the clustering based compression algorithms, and the Bloom Filter based string recovery algorithm. After that, the effectiveness of the proposed method in misconfiguration avoidance will be evaluated with real and synthetic configuration files. All algorithms are implemented in C++ and tested on Ubuntu 16.04.6 LTS with Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz processor and 16GB memory.

A. String Clustering Evaluation

The experimental results of the set cover based and VP-Tree based clustering algorithms are shown in Table III and IV respectively. In these two tables, the time refers to the time required to process all strings in each dataset, and the compression ratio is defined as the number of uncompressed strings (including centroids and outliers) divided by the total number of original strings. And the clustering radius r' is set to 5, which is the maximum value allowed in the experiment computer (otherwise, it will take too much time for recoverBF to recover the compressed strings). From the results we can see that the compression ratio of set cover based algorithm is much less than that of VP-Tree based algorithm. However, it takes much more time for the set cover based algorithm to cluster all strings in each dataset than that of VP-Tree based algorithm. It is worth noting that almost all strings in the Name dataset is compressed by the set cover based algorithm, only 0.10% of all strings are left. However, the major disadvantage is it takes too much time (29370 seconds, more than 8 hours). In comparison, the VP-Tree based algorithm delivers a low enough compression ratio (3.93%), while the processing time is relatively acceptable, only 118 seconds. Generally speaking, set cover based algorithm delivers a better compression ratio, while the VP-Tree based algorithm takes much shorter processing time which is consistent with the theoretical analysis.

TABLE III
RESULTS OF CLUSTERING ALGORITHM BASED ON SET COVER

Dataset	Time/seconds	Compression ratio
Word	115	4.02%
Name	29370	0.10%
Pwd	1962	2.68%

TABLE IV
RESULTS OF CLUSTERING ALGORITHM BASED ON VP-TREE

Dataset	Time/seconds	Compression ratio
Word	14	35.99%
Name	118	3.93%
Pwd	163	36.00%

B. String Recovery Evaluation

The string recovery algorithm delivers an exponential time complexity. Here we mainly check whether or not it can be used to recover all the strings compressed by the

*https://www.kaggle.com/datagov/usa-names
†http://datashaping.com/passwords.txt

TABLE V
CONFIGURATION FILE DETAILS

Configuration	Total configuration pairs	Error configuration pairs
IACS	30000	3000
Apache	20000	2000
MySQL	10000	1000
CentOS	50000	5000

TABLE VI
QUERY RESULT

Configuration	Total queries	Successful queries (τ = 5)	Successful queries (τ = 10)
IACS	3000	2410	3000
Apache	2000	1685	2000
MySQL	1000	812	1000
CentOS	5000	4122	5000

clustering algorithms in subsection IV-A. To be precise, the recovery algorithm takes the centroids returned by the clustering algorithm as the query strings. And then we check each string generated by the recovery algorithm whether it is in the respective dataset (Word, Name and Pwd) or not. After throughout verifying all the generated strings, we find that there are no omissions for all three datasets, i.e., all the strings compressed by the clustering algorithms can be recovered.

C. Misconfiguration Avoidance Evaluation

In this subsection, we use the real configuration files from IACS as well as some famous open source systems (Apache, MySQL and CentOS) to check whether the proposed method can be used to avoid the misconfiguration effectively. The IACS dataset consists of the configuration files from multiple systems including distributed control system (DCS), supervisory control and data acquisition (SCADA), and various IIoT devices. CentOS is an enterprise-class Linux distribution, MySQL is a database server, Apache is a web server. The CentOS, MySQL and Apache datasets contain the configuration files from the respective system. For the convenience of evaluation, all the configuration files of IACS are combined into one, referred as "IACS configuration files". The numbers of pairs of <configuration directive, value> in these configuration files are very small, e.g., no more than 500 in the original Apache configuration file. In order to simulate the real configuration changing scenario, we generate much more configuration pairs (i.e., configuration directive with its value) with the following method: 1) for each configuration directive, change its respective values; 2) for each pair of <configuration directive, value>, make some typos (less than 10 characters) in the configuration directive as well as its value. As a result, the details of the configuration files are illustrated in Table V, where the error configuration pair means the pair has unacceptable values or typos.

Followed that, the effectiveness of the proposed method in aiding the configuration management will be evaluated with the following steps. First of all, all the configuration pairs in each configuration file are processed by the proposed method. And then, for each error configuration pair, a query is carried out to find all the similar pairs within the edit distance 5 and 10 respectively. If returning some correct configuration pairs which can be used to correct the error configuration, then this query is believed to be successful. Otherwise, this query is considered to be unsuccessful. The experiment results are depicted in Table VI. From this table we can see that since we limit the maximum misspelled characters (less than 10), all the queries are successful when the edit distance $\tau = 10$. There really exist some unsuccessful queries when $\tau = 5$, this is because that there are some error configuration pairs which have more than 5 misspelled characters. In summary, the proposed method can effectively help correcting the misconfiguration.

V. CONCLUSION

This paper proposes a misconfiguration avoidance method for IACS based on streaming string similarity search. Two clustering-based string compression algorithms are proposed to keep the strings stored in memory as many as possible. The set cover based clustering algorithm delivers higher compression efficiency, while the VP-Tree based clustering algorithm exhibits a higher compression speed. At the same time, a historical string recovery algorithm based on Bloom Filter is proposed, so that the deleted historical strings can be recovered without omission when a query arrives. The major drawback is the low string recovery speed which can be further improved by parallelization. The experiments show that the proposed method can effectively help correcting the misconfigurations of IACS. In the future work we plan to accelerate the string recovery algorithm on the GPU using CUDA.

REFERENCES

[1] Q. Zhang, C. Zhou, Y.-C. Tian, N. Xiong, Y. Qin, and B. Hu, "A fuzzy probability bayesian network approach for dynamic cybersecurity risk assessment in industrial control systems," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 6, pp. 2497–2506, 2017.

[2] X. Li, C. Zhou, Y.-C. Tian, and Y. Qin, "A dynamic decision-making approach for intrusion response in industrial control systems," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 5, pp. 2544–2554, 2018.

[3] F. Zhang, H. A. D. E. Kodituwakku, J. W. Hines, and J. Coble, "Multilayer data-driven cyber-attack detection system for industrial control systems based on network, system, and process data," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 7, pp. 4362–4369, 2019.

[4] C. Zhou, X. Li, S.-H. Yang, and Y.-C. Tian, "Risk-based scheduling of security tasks in industrial control systems with consideration of safety," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 5, pp. 3112–3123, 2020.

[5] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: Finding the needle in the haystack," in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, vol. 6, 2004, pp. 77–90.

[6] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: improving configuration management with operating system causality analysis," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 237–250, 2007.

[7] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure." in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, vol. 4, 2004, pp. 245–257.

[8] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in internet services." in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, vol. 4, 2004, pp. 61–76.

[9] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and validating database system administration." in *USENIX Annual Technical Conference, General Track*. Boston, MA, 2006, pp. 213–228.

[10] P. Anderson, P. Goldsack, and J. Paterson, "Smartfrog meets lcfg: Autonomous reconfiguration with central policy control." in *LISA*, vol. 3, 2003, pp. 213–222.
[11] W. Zheng, R. Bianchini, and T. D. Nguyen, "Automatic configuration of internet services," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007, pp. 219–229.
[12] Y. Jiang, G. Li, J. Feng, and W.-S. Li, "String similarity joins: An experimental evaluation," *Proc. VLDB Endow.*, vol. 7, no. 8, p. 625–636, Apr. 2014.
[13] S. Wandelt, D. Deng, S. Gerdjikov, S. Mishra, P. Mitankin, M. Patil, E. Siragusa, A. Tiskin, W. Wang, J. Wang *et al.*, "State-of-the-art in string similarity search and join," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 64–76, 2014.
[14] L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen, "Efficient metric indexing for similarity search and similarity joins," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 3, pp. 556–571, March 2017.
[15] W.-K. Hon, R. Shah, and J. S. Vitter, "Compression, indexing, and retrieval for massive string data," in *Proceedings of the 21st Annual Conference on Combinatorial Pattern Matching*. Berlin, Heidelberg: Springer-Verlag, 2010, p. 260–274.
[16] L. Chen, Y. Gao, B. Zheng, C. S. Jensen, H. Yang, and K. Yang, "Pivot-based metric indexing," *Proc. VLDB Endow.*, vol. 10, no. 10, p. 1058–1069, Jun. 2017.
[17] L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen, "Efficient metric indexing for similarity search," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 591–602.
[18] J. Cui, W. Wang, D. Meng, and Z. Liu, "Continuous similarity join on data streams," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2014, pp. 552–559.
[19] G. D. F. Morales and A. Gionis, "Streaming similarity self-join," *arXiv preprint arXiv:1601.04814*, 2016.
[20] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," *Computer Networks*, vol. 57, no. 18, pp. 4047 – 4064, 2013.
[21] G. Li, D. Deng, J. Wang, and J. Feng, "Pass-join: A partition-based method for similarity joins," *Proc. VLDB Endow.*, vol. 5, no. 3, p. 253–264, 2011.
[22] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120–133, 2010.
[23] N. Alon, B. Awerbuch, and Y. Azar, "The online set cover problem," in *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, 2003, p. 100–105.
[24] P. Slavík, "A tight analysis of the greedy algorithm for set cover," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, 1996, p. 435–441.
[25] G. Cormode, H. Karloff, and A. Wirth, "Set cover algorithms for very large datasets," in *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, 2010, p. 479–488.
[26] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Soda*, vol. 93, no. 194, 1993, pp. 311–321.

Yu Zhang received the B.E. degree in computer science and the Ph.D. degree in computer system architecture from the Harbin Institute of Technology, Harbin, China, in 2004 and 2010, respectively. After graduation, he joined the College of Computer and Control Engineering, Nankai University, Tianjin, China. He is currently an Associate Professor with the College of Cyber Science, Nankai University. He has authored or coauthored more than 30 academic papers in international conferences and journals. His research interests include machine learning, data mining, artificial intelligence security and network security, particularly cyberspace security situational awareness.

Yani Ge received the B.E. degree in information security, in 2019, from the China University of Geosciences, Wuhan, China. She is currently working toward the M.E. degree in computer science at Nankai University, Tianjin, China. Her research interests include network security and data mining.

Peiran Yu received the B.E. degree in information security, in 2019, from the Nankai University, Tianjin, China. She is currently working toward the M.E. degree in computer science at Nankai University, Tianjin, China. Her research interests include network security and data mining.

Jianzhong Zhang received the Ph.D. degree in computer science from the Nankai University, Tianjin, China, in 2008. He is currently a Professor and Ph.D. Supervisor with the College of Cyber Science, Nankai University. In recent years, he has authored or coauthored more than 50 academic papers in international conferences and journals. He has cultivated more than 30 graduate students and 6 doctoral students, published 11 textbooks related to computer networks. He won the first prize of Tianjin Teaching Achievement in 2009 and the first prize of Nankai University Teaching Achievement in 2009 and 2017 respectively. His research interests include mobile computing, activity recognition, and network security.

Yongzheng Zhang (M'13) received the B.S. and Ph.D. degrees in computer science from the Harbin Institute of Technology, Harbin, China, in 2001 and 2006, respectively. He is a Professor and Ph.D. Supervisor with the Institute of Information Engineering, Chinese Academy of Sciences (CAS), Beijing, China. His research interests include network security, particularly cyberspace security situational awareness. Prof. Zhang was honored with the first prize of the Chinese National Award for Science and Technology Progress in 2011.

Thar Baker is a Senior Lecturer in Software Systems in the Department of Computer Science at the Faculty of Engineering and Technology. He has received his Ph.D. in Autonomic Cloud Applications from LJMU in 2010. His research interests include: Cloud Computing, Distributed Software Systems, Big Data, Algorithm Design, Green and Sustainable Computing, and Autonomic Web Science. He has been actively involved as member of editorial board and review committee for a number peer reviewed international journals, and is on programme committee for a number of international conferences. Dr. Baker was appointed as Expert Evaluater in the European FP7 Connected Communities CONFINE project (2012-2015). He worked as Lecturer in the Department of Computer Science at Manchester Metropolitan University (MMU) in 2011.