Creating Character-based Templates for Log Data to Enable Security Event Classification

Markus Wurzenberger, Georg Höld,
Max Landauer, Florian Skopik
firstname.lastname@ait.ac.at
AIT - Austrian Institute of Technology
Vienna, Austria

Wolfgang Kastner
wolfgang.kastner@tuwien.ac.at
Vienna University of Technology
Vienna, Austria

ABSTRACT

Log data analysis is an essential task when it comes to understanding a computer's or a network's system behavior, and enables security analysis, fault diagnosis, performance analysis, or intrusion detection. An established technique for log analysis is log line clustering, which allows to group similar events and to detect outliers, malicious clusters or changes in system behavior. However, log line clusters usually lack meaningful descriptions that are required to understand the information provided by log lines within a cluster. Template generators allow to produce such descriptions in form of patterns that match all log lines within a cluster and therefore describe the common features of the lines. Current approaches only allow generation of token-based (e.g., space-separated words) templates, which are often inaccurate, because they do not recognize words that can be spelled differently as similar and require further information on the structure and syntax of the data, such as predefined delimiters. Consequently, novel character-based template generators are required that provide robust templates for any type of computer log data, which can be applied in security information and event management (SIEM) solutions, for continuous auditing, quality inspection and control. In this paper, we propose a novel approach for computing character-based templates, which combines comparison-based methods and heuristics. To achieve this goal, we solve the problem of efficiently calculating a multi-line alignment for a group of log lines and compute an accurate approximation of the optimal character-based template, while reducing the runtime from $O(n^m)$ to $O(mn^2)$. We demonstrate the accuracy of our approach in a detailed evaluation, applying a newly introduced measure for accuracy, the Sim-Score, which can be computed independently from a ground truth, and the established F-Score. Furthermore, we assess the robustness of the algorithm and the influence of different log data properties on the quality of the resulting templates.

CCS CONCEPTS

• **Security and privacy → Intrusion detection systems**; *Systems security*; *Network security*; • **Computing methodologies →** Machine learning;

KEYWORDS

template generation; character-based templates; log analysis; multi-line alignment

ACM Reference Format:
Markus Wurzenberger, Georg Höld, Max Landauer, Florian Skopik and Wolfgang Kastner. 2020. Creating Character-based Templates for Log Data to Enable Security Event Classification. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3320269.3384722

ASIA CCS '20, October 5–9, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/10.
https://doi.org/10.1145/3320269.3384722

1 INTRODUCTION

Grouping log lines using clustering and classification algorithms is an established method to analyze a computer networks' log data. Clustering is also the basis of further analysis methods, such as outlier detection [12] and time series analysis [6], which are often applied in cyber security and threat detection. These methods allow to detect suspicious anomalous events and changes in network behavior which are consequence of malicious misuse caused by attackers and malware or erratic behavior initiated by misconfiguration and faulty usage. Once log data are clustered, it is possible to statistically describe these clusters' properties, such as size, or diameter. However, most clustering algorithms provide no or only inaccurate and insufficient information on the content of a log line cluster. Thus, template generators are required that allow to generate meaningful cluster descriptions. Additionally, templates support the process of generating log parsers [2]. Numerous security applications benefit from templates and template generators, including security information and event management (SIEM) solutions, intrusion detection systems (IDS), parser and signature generators. Furthermore, templates can be applied for log classification in general, for log reduction through filtering, and for event counting.

A template is basically a string that consists of substrings which occur in each log line of a cluster in a similar location. Those substrings are referred to as static parts of the log lines of the cluster. They are separated by wild cards, which represent variable parts of the log lines, such as usernames, IP addresses, and identifiers (ID). Furthermore, a template has to match all log lines of the corresponding cluster.

The unsolved problem of generating a sequence alignment for more than two log lines, i.e., generating a multi-line alignment, is one of the main reasons why currently existing template generators follow token-based approaches and not character-based ones. In this context, tokens are substrings of a string, separated by a predefined delimiter, e.g., space or comma. Token-based template

generators first split log lines into tokens. Afterwards, they generate a template, where tokens that represent static parts of the log lines, i.e., occur in all log lines in the same location, remain part of the template, and all other tokens are replaced by wild cards. The biggest advantage of token-based template generators is their high performance with respect to runtime. However, this procedure leads to some significant drawbacks. Token-based template generators prevent that tokens corresponding to substrings with high similarity, which only differ in a few symbols, become part of a template. Thus, they consider words and terms that can be spelled differently, such as `php-admin`, `PHP-Admin` and `phpadmin`, or when SQL queries are used, `username` and `u.username`, as completely different. Furthermore, those approaches require a predefined list of delimiters, which strongly depends on the present log data. Moreover, due to the token-based approach, larger parts of log lines are covered by wild cards, since tokens are considered entirely different, even if they only vary in a single symbol. Additionally, it is often not clear how many tokens a single wild card represents. Most of the times, a single wild card replaces a different number of tokens, depending on the log lines that match the template.

In contrast to token-based template generators, character-based approaches do not rely on predefined building blocks in form of tokens. These approaches recognize static and variable parts of log lines independently from predefined delimiters. Figure 1 provides an example for the two different types of templates (assuming spaces as delimiters for the token-based approach) for a certain cluster.

In this paper, we propose an approach for generating character-based templates to overcome the disadvantages of token-based approaches. The main challenge to achieve this goal is to calculate a multi-line sequence alignment [9], i.e., a sequence alignment for more than two lines. A sequence alignment arranges two character sequences by aligning their identical or similar parts and recognizing optional and variable characters. There exist many efficient algorithms and string metrics to achieve this for two character sequences [12]. Furthermore, there are algorithms for genetic or biologic sequences to calculate pair-wise and multi-line alignments, which however require knowledge about the evolution of nucleotides and are therefore not suitable for log data [9]. Algorithms to align multiple sequences of any characters with no evolutionary context are still missing. The main reason is the difficulty to overcome the high computational complexity of this problem, which is at least $O(n^m)$, where n is the length of the shortest log line and m is the number of lines in a cluster.

Hence, we propose a character-based cluster template generator that incrementally processes the lines of a log line cluster and reduces the computational complextiy $O(n^m)$ to $O(mn^2)$. The main contributions of the paper are:

(i) Four algorithms to compute multi-line sequence alignments for any strings.
(ii) An incremental approach to efficiently generate character-based templates that provide a more detailed representation than token-based templates.
(iii) A universally applicable template generator for log data independent from delimiters.
(iv) A template generator that overcomes the problem of too generic or over-fitting templates.
(v) Evaluation of the accuracy of the proposed algorithms, as well as qualitative and quantitative comparison to token-based approaches carried out on real data.

The remainder structures as follows: Section 2 summarizes background and related work. The concept of the approach is introduced in Sec. 3 and Sec. 4 describes the different algorithms for generating character-based templates in detail. Finally, Sec. 5 evaluates and compares the algorithms, and Sec. 6 concludes the paper and describes future work.

2 BACKGROUND AND RELATED WORK

Currently, most template generators follow token-based approaches. Many of them build on clustering [5]. For example, SLCT [10] follows a density-based clustering approach. Thereby, frequent words on certain positions in the log lines are considered as static and infrequent ones as variable. IPLoM [8], on the other hand, implements partitioning. Hence, log lines are split at appropriate token positions and sorted into subgroups iteratively. Furthermore, many log parser generators provide token-based templates or build on template generators [2]. Two examples for tree-based approaches are Drain [3] and AECID-PG [11], which depict log data as graph-theoretical tree, where each node represents a token with an either static or variable pattern. Following the branches of a parser tree allows to obtain log templates.

The foundations for character-based templates are string metrics that allow to compare two strings character-wise. Some well-known examples for such string metrics are the Levenshtein distance, the Needleman-Wunsch algorithm, the Smith-Waterman algorithm, and different versions of the Jaro distance [1]. In this paper, we focus on the Levenshtein distance, which counts the edit operations that are required to transform a string into another one. By reverting this procedure and leveraging the computed score-matrix, it is possible to calculate an alignment. Other algorithms, such as the Needleman-Wunsch and Smith-Waterman, provide an alignment at once, but suffer from a higher computational complexity due to a more complex scoring function. However, all these algorithms are only able to provide pairwise sequence alignments.

In the area of bioinformatics, there exist several highly efficient algorithms, such as MAFFT, M-Coffee and PROMALS, that allow to compute so-called multiple sequence alignments [9]. These algorithms mostly base on previously mentioned methods for calculating pairwise sequence alignments. Due to the fact that they use scoring systems and heuristics that make use of evolutionary relationships between amino acids, they can only be applied to strings that represent biological sequences such as DNA and RNA, and not to any other type of string [9]. Therefore, efficiently generating a template for a group of similar strings remains an unsolved problem. Furthermore, it is not expedient to calculate the optimal alignment for a group of strings, because it would be computationally too expensive. Hence, it is only feasible to approximate the optimal template.

```
Cluster:
database-1.server.d3.local mysql-normal ORDER BY status-system
database-0.server.d4.local mysql-normal GROUP BY status-network
database-1.server.d3.local mysql-normal GROUP BY status-system
database-0.server.d4.local mysql-normal ORDER BY status-network

Template token-based:
[*] mysql-normal [*] BY [*]

Template character-based:
database-[*].server.d[*].local mysql-normal [*]R[*] BY status-[*]t[*]
```

Figure 1: Example of templates for a cluster of SQL logs.

3 CONCEPT

In the following, we describe a novel concept that allows to efficiently generate character-based templates for groups of similar log lines, e.g., pre-clustered log lines. The goal of computing a template for a group of log lines is to recognize static and variable parts occurring in all of the lines. This allows to determine shared properties and enables the design of meaningful log line cluster descriptions in form of templates that can be used for further analysis. Since the aim is to recognize common properties, templates are generated for log lines that reach a certain similarity, because otherwise a template would not provide any benefit.

In the remainder, the term template always refers to character-based templates. Furthermore, we define the template of a log line cluster as an ordered list of substrings that occur in the same order in each log line of the cluster. In case of the given example in Fig. 1, the template would be [`database-`, `.server.d`, `.local mysql-normal`, `R`, `BY status-`, `t`]. The example shows that for the words `ORDER` and `GROUP` only the character `R` remains part of the template. While there exist several solutions to determine a template for two log lines, it is not trivial to efficiently compute the optimal template for a group of log lines. For two log lines, the template can be generated by simply calculating the pairwise string alignment applying, for example, the Levenshtein (LV) distance or the Needleman-Wunsch algorithm. On the contrary, generating a template for a group of log lines, a so-called multi-line alignment, is complicated. The computational complexity to calculate the optimal template for a group of log lines, applying comparison-based algorithms that omit any heuristics, cannot be lower than $O(n^m)$, where n is the length of the shortest log line within a cluster and m is the number of lines in a cluster. The computational complexity is that high, because each line of a cluster has to be compared with each other line. Due to the large amount of log data, which template generators might have to process, both n and m can be large, which results in a long runtime. On the opposite, for token-based template generators this is not such an issue, because n then refers to the number of tokens within the log lines, which is much smaller than the number of characters. Thus, the goal of the approach we propose is to efficiently compute an approximation of the optimal template for a group of log lines, where each log line of the cluster has to be processed only once.

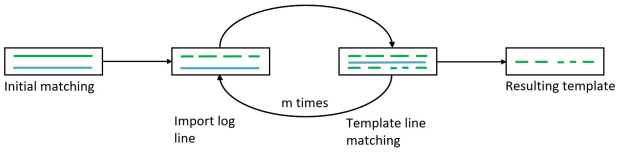


Figure 2: Template generation process flow.

The approach we propose significantly reduces the computational complexity of computing a character-based log cluster template. Figure 2 illustrates the process flow for generating templates for log line clusters. The algorithm processes log lines sequentially and thus follows an incremental approach, which has to handle each line only once. In each step, the algorithm adapts the template. In the following, the term *current template* refers to these temporary templates. Initially, the first line of a cluster defines the current template for the cluster. Next, the algorithm calculates the pairwise alignment between the initial template, i.e., the first line of the cluster, and the second line of the cluster. Afterwards, the algorithm compares the current template with each remaining line in the cluster and adapts the template accordingly. In order to efficiently accomplish this adaptation, we propose four different procedures for this task and compare their advantages and disadvantages. The runtime of these algorithms mainly depends on the applied distance. Our approach uses the LV-distance, because of its relatively low computational complexity of $O(n^2)$, compared to other string metrics that can be applied for calculating pairwise alignments. Hence, it is possible to process a cluster in less than $O(mn^2)$ runtime, where n is the length of the longest line, which takes the most time to be processed and m is the number of lines in the cluster. Furthermore, it is possible to modify these algorithms by replacing the LV-distance with any other string metric that allows to calculate an alignment. Since the input data is pre-clustered, the resulting template has a high similarity to the optimal template, as shown in the evaluation presented in Sec. 5 by calculating two different metrics that measure the accuracy of the algorithms.

4 CLUSTER TEMPLATE GENERATOR ALGORITHMS

This section introduces four different algorithms to generate chracter-based templates for pre-clustered log data. The first two algorithms follow quite different approaches, while the third one combines the advantages of both and simultaneously mitigates their disadvantages. The fourth algorithm combines the token-based and character-based approach. All proposed algorithms build on the calculation of pairwise string alignments, which leverages string metrics. In this paper, we focus on the Levenshtein-distance (LV-distance). It is possible to replace the LV-distance by any other distance, which allows to return the shared substrings of two compared strings. We also experimented with the Needleman-Wunsch-distance, but in comparison to the LV-distance the runtime is significantly higher for an output of comparable quality.

The remaining section first describes the initial matching between the initial template, i.e., the first processed log line, which is the one with the earliest timestamp, unless otherwise stated, and the second line of a log cluster, which is the one with the second earliest timestamp. This step is identical for all four algorithms. Afterwards, we define the three purely character-based algorithms *merge*, *length* and *equalmerge*, which enable matching a template with a log line. Thus, they incrementally process all log lines of a log cluster in temporal order to sequentially refine the template, so that the resulting template matches all log lines of the cluster. Finally, we introduce the *token_char* algorithm which combines the token-based and character-based approach to calculate character-based templates.

4.1 Initial matching

Since a template is defined as a list of substrings that occur in the same order in each log line of a cluster, a string-list characterizes each template. In the following, the term *block* refers to these strings.

Initially, the first template is equivalent to the temporal first line of the cluster. Thus, the string-list consists of a single string which is equal to the first log line of the cluster. Next, the algorithm calculates the LV-distance between the initial template, which is a string, and the second log line of the cluster. The string-list of the template, which is equal to the first line, is now adapted to the substrings shared with the second line according to the LV-distance.

Figure 3 illustrates how the first matching of log lines is accomplished. The green blocks represent the template before and after the matching, and the blue block corresponds to the log line which the current template is matched to. Additionally, Alg. 1 describes the implementation of the initial matching, which is a combination of the calculation of the LV-distance between two strings and a modification of the commonly used backtrace procedure to compute the alignment of two strings based on the resulting scoring matrix of the LV-distance calculation [4]. The algorithm described in Alg. 1 takes as input the scoring matrix of the LV-distance M and the *path* in M that relates to the optimal alignment. For this, the path is represented by the list of directions that have to be taken through the scoring matrix during the backtrace procedure. In the for loop, the algorithm extends the currently generated substring with the currently processed character if the direction is *diagonal*

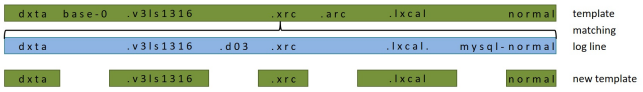


Figure 3: Initial matching.

and the compared strings have equal characters at the compared position[1]. It ends the substring and appends an empty string to list T, which represents the template, if the direction is *up* or *down*. The latter is only done, if the last element of the list $\text{LAST}(T)$ is not an empty string. In the returned list of substrings T, empty strings represent gaps, which are defined as wildcards for the text between two blocks of a template.

Algorithm 1 $\text{STRING\_STRING\_MATCHING}(S_1, S_2)$

```
1:  M ← LV_MATRIX(S1, S2);
2:  path ← Path in M from M[0][0] to M[LEN(S1)][LEN(S2)];
3:  T ← [''];
4:  x ← 0;
5:  y ← 0;
6:  for directions ∈ path do
7:      if direction = diagonal then
8:          x ← x + 1;
9:          y ← y + 1;
10:         if S1[x] = S2[y] then
11:             LAST(T).APPEND(S1[x]);
12:         else
13:             T.APPEND('');
14:         end if
15:     else if direction = down then
16:         x ← x + 1;
17:         if LAST(T)! = '' then
18:             T.APPEND('');
19:         end if
20:     else if direction = right then
21:         y ← y + 1;
22:         if LAST(T)! = '' then
23:             T.APPEND('');
24:         end if
25:     end if
26: end for
27: return T
```

4.2 Merge algorithm

The merge algorithm is the most straightforward of the considered algorithms. Figure 4 depicts the matching between a template and a log line. First, the algorithm converts the template into a single string by merging the blocks together, i.e., by concatenating the strings in the list into a single string. Then, the LV-distance between this aggregated string and the log line is calculated. Thus, the previous template is adapted, according to the LV-distance, so that it matches also the new log line. Note, it is prohibited that the algorithm deletes already existing gaps in the template, because if

[1]Note, the direction is also diagonal when a character should be replaced.

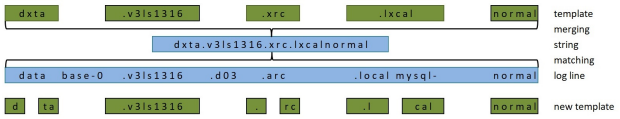


Figure 4: Merge algorithm matching: The green blocks represent the template, the upper blue block the merged template and the lower blue block the log line.

this happens the template does not fit previously processed lines anymore. However, gaps are not considered as mandatory, i.e. they do not have to occur in all lines. Algorithm 2 describes the linear procedure consisting of three steps: (i) merge the current template T_1 to a single string S_1, (ii) use Alg. 1 to compute the alignment T_2 between the merged template S_1 and the log line S_2, and (iii) ensure that no gaps that existed in the previous template T_1 are missing in the resulting template T.

Algorithm 2 MERGE(T_1, S_2)

```
1: S_1 ← MERGE_TEMPLATE _TO_STRING(T_1);
2: T_2 ← STRING_STRING_MATCHING(S_1, S_2);
3: T ← ALIGN_GAPS(T_1, T_2);
4: return T
```

4.3 Length algorithm

The merge algorithm always calculates the LV-distance for a log line and the current template, which results in a rather long runtime. Hence, the length algorithm instead only calculates the LV-distance for blocks and corresponding substrings of the log line. This reduces the runtime, because the length of the strings, for which the algorithm calculates the LV-distance, is shorter.

The length algorithm processes the blocks in order of their lengths, beginning with the longest one. Since the algorithm does not process the blocks from left to right and calculates the LV-distance between blocks of the template and corresponding substrings of the log line, it first has to localize which block corresponds to which part of the log line. The localization process is described in more details later in this section. Processing the blocks in order of their length prohibits that smaller blocks prevent larger ones from becoming part of the new template, or to force the algorithm to split them. Therefore, the template tends to include more characters which results in a higher coverage, i.e., on average more characters of the log lines are part of the template of the corresponding cluster. Furthermore, longer blocks are considered more significant for a cluster than shorter ones.

Figure 5 supports the description of the length algorithm. The algorithm starts with the localization procedure. For that purpose, it marks all blocks of the template that occur as identical substrings in the log line, starting with the longest one. Figure 5 depicts this in the first two lines by connecting block 1 and 3 with equal substrings in the template. During the marking process, the algorithm does not consider the whole line for all blocks, but only a valid section to sustain the order of the blocks. For example, the second processed block `.lxcal` in Fig 5, can only mark a substring in the section `.d03.arc.local.mysql-normal`, because it has marked blocks to the left and to the right. Empirical studies support to only consider blocks consisting of more than two characters in this phase to avoid that larger blocks are excluded from the resulting template. This leads to templates of higher quality.

Once the algorithm marked all blocks of the template that identically occur as substrings in the log line, it processes the remaining blocks of the template, again in the order of their lengths starting with the longest. Lines three to five in Fig. 5 visualize this procedure. Each unmarked block of the template is matched with the corresponding section of the log line. As Fig. 5 illustrates, the matched block gets either split or deleted according to the LV-distance. After the matching, the substring that matched the block becomes a marked section and is not further considered in the matching process. For example, the algorithm matches the first processed block `.lxcal` in the lower part of Fig 5 with the corresponding substring `.local`. Thus, the algorithm marks this substring, which is illustrated by the dashed rectangle. Therefore, the algorithm matches the third block with a shorter section than the first block.

Note, if at any point during this procedure two blocks have the same size, the algorithm processes the leftmost one first. The fact that similar log lines usually differ more from each other towards the end, supports this decision. As Alg. 3 demonstrates, in opposite to the merge algorithm, the input template is modified and returned and not generated from scratch. Therefore, the gap alignment can be omitted. The length algorithm consists of two for loops, one for the marking process and a second one that matches unmarked blocks. Hence, Alg. 3 applies Alg. 1 to match all blocks (*str* in the Alg. 3) from the current template T_1, that have not been marked yet, to corresponding substrings in log line S_2. Once a substring of S_2 has been matched, it is marked so that no other block of T_1 can be matched to it. Algorithm 4 describes the function `CORRESPONDING_SUBSTR`. It returns for a block of the template $T[j]$ the corresponding substring in log line S. Note, if there is no corresponding substring, the algorithm returns an empty string.

Algorithm 3 LENGTH(T_1, S_2)

```
1: for str ∈ T_1 ordered by length do
2:     if str ⊆ CORRESPONDING_SUBSTR(S_2, str) then
3:         Mark str in T_1 and S_2;
4:     end if
5: end for
6: for unmarked str ∈ T_1 ordered by length do
7:     replace str with STRING_STRING_MATCHING
       (str, CORRESPONDING_SUBSTR(S_2, str));
8:     Mark the matched string in S_2;
9: end for
10: return T_1
```

Because of the marking procedure of the length algorithm, the algorithm has to calculate the LV-distance only for the remaining unmarked blocks. Therefore, the runtime of the length algorithm is significantly lower than the runtime of the merge algorithm, which calculates the LV-distance for the whole template and log line. Since the log lines are considered pre-clustered, they have a high similarity, which means that the the marking process significantly

Algorithm 4 CORRESPONDING_SUBSTRING($S, T[i]$)

if ∃ marked block $T[j]$ in T, with $j < i$ **then**
  $j \leftarrow$ Next smaller index of a marked block in T;
  $m \leftarrow$ Index of last marked character of $T[j]$ in S;
else
  $m \leftarrow 0$;
end if
if ∃ marked block $T[k]$ in T, with $k > i$ **then**
  $k \leftarrow$ Next higher index of a marked block in T;
  $n \leftarrow$ Index of first marked character of $T[k]$ in S;
else
  $n \leftarrow$ LEN(S);
end if
return $S[m, n]$

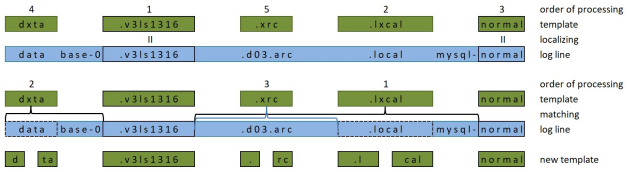


Figure 5: Length algorithm marking and matching: The green blocks represent the template and the blue blocks the log line. The upper part illustrates the marking. The lower part visualizes the matching of the remaining blocks. The horizontal brackets highlight the sections of the log line which are matched with the blocks. The dashed rectangle in the lower blue block represents the marked section which originates from the matching with block 1 from above.

reduces the runtime. However, while the marking process reduces the runtime, it might also reduce the quality of the template, because the matching is optimized with respect to sections within the strings and not globally over the whole strings. The evaluation discusses this in Sec. 5.7.

4.4 Equalmerge algorithm

Figure 6 depicts the matching between a template and a log line applying the equalmerge algorithm. The following algorithm combines the features of the merge and the length algorithm. Equally to the length algorithm, the equalmerge algorithm first marks the blocks, which occur as substrings in the log line. After the marking, the algorithm merges the blocks remaining between the marked blocks of the template identical to the merge algorithm. The algorithm merges the unmarked blocks according to their corresponding section. Hence, for example, it merges in line three of Fig. 6 the remaining unmarked blocks between block 1 and block 2 from line one to a single block. Finally, the newly created blocks are matched with the related sections of the log line. These blocks are split or gaps are included according to the LV-distance. Equally to the merge algorithm, it is prohibited that the algorithm deletes gaps.

Algorithm 5 and Alg. 3 show that the implementations of the equalmerge and the length algorithm are similar to each other and

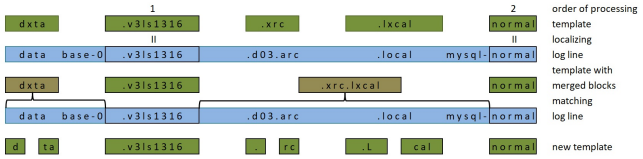


Figure 6: Equalmerge algorithm matching.

differ only in the second for loop. In the second for loop of the equalmerge algorithm adjacent unmarked strings, i.e. unmarked strings between marked stings, are aggregated to *adj_strings*. Afterwards, Alg. 2 is applied to compute the alignment (T_3) between *adj_strings* and the corresponding substring of the log line S_2. Finally, alignment T_3 replaces the strings in the current template T_1 that have been aggregated to *adj_strings*.

Algorithm 5 EQUALMERGE(T_1, S_2)

for $str \in T_1$ ordered by length **do**
  if $str \subseteq$ CORRESPONDING_SUBSTR($S_2, string$) **then**
    Mark str in T_1 and S_2;
  end if
end for
for unmarked $str \in T_1$ **do**
  $adj\_strings \leftarrow$ adjacent unmarked strings of str in T_1 including str itself;
  $T_3 \leftarrow$ MERGE($adj\_strings$, CORRESPONDING_SUBSTR(S_2, str));
  Replace $adj\_strings$ in T_1 with T_3;
  Mark T_3 and the matched string in S_2;
end for
return T_1

The equalmerge algorithm implements a refinement of the length algorithm. Since it calculates the LV-distance between the merged blocks of the template and the corresponding substring of the log line, it has a slightly longer runtime than the length algorithm, but simultaneously the resulting template inherits the higher quality of the merge algorithm. At the same time, the runtime of the equalmerge algorithm is shorter than the one of the merge algorithm, while the decrease of the quality of the template is smaller than the one of the length algorithm.

4.5 Token_char algorithm

Since most template generators operate token-based, we developed a hybrid approach, which should combine the advantages of both token-based and character-based approaches. While, for example, token-based templates are easier to convert into parser models, character-based templates provide a more detailed description of log line clusters and provide more accurate signatures. Thus, to accomplish a hybrid template, we separate the template into two layers. The first layer comprises the token-structure, which contains the token-list that stores the tokens. The second layer composes the character-structure. Therefore, a character-structure is assigned to each gap, which contains a character-based template for the tokens that are replaced by the gap. In the end, the token and the character structure are merged to a character-based template.

Figure 7 depicts the procedure of the matching performed by the token_char algorithm and supports the algorithm's description. The initial step of the token_char algorithm differs from the previous algorithms. First, the algorithm converts all log lines of a cluster into token-structures, i.e., lists of tokens. Therefore, the algorithm splits the log lines into substrings at predefined delimiters. Hence, this algorithm inherits the disadvantage of token-based template generators, which have to split all log lines at the same delimiters, whether it is useful or not. Next, between each token, a character-structure is established which initially contains the corresponding delimiter. Finally, the token-char-structure of the temporal first log line represents the initial template.

The following describes the matching procedure between a token-char-template and the token-char-structure of a log line. In the first step, the algorithm matches the two token-structures. Therefore, the algorithm searches for tokens in the log line's token-structure that correspond to the tokens in the token-structure of the template. The distance metric the algorithm uses is a modification of the LV-distance, which treats tokens like characters and weights their value for the accuracy of the template by their length. This is necessary, because the normal LV-distance applied to token-structures would provide the template with the most tokens, without taking into account that a token consisting of a larger number of characters supports a template with higher coverage. Otherwise, a template with low coverage would be accepted as long as it consists of a large number of tokens. Thus, our algorithm matches the tokens according to the LV-distance with the difference, if two tokens of the template match the same corresponding token of the log line's token-structure, the score assigned by the algorithm for computing the LV-distance is decreased by the length of the token. This is reasonable, because when calculating the LV-distance, positive scores represent penalties, i.e., positive values correspond to required modification operations when transforming one string into another. Note, the result is not a distance, but a sufficient score for this algorithm. The first two lines of Fig. 7 depict the matching of the token-structures.

Next, the algorithm converts the tokens of the token-structure of the template which do not match any of the log line's into character-structures and merges all adjacent character-structures. Hence, there exists exactly one character-structure between two tokens as line 3 of Fig. 7 shows. Finally, the char-structures of the current template are matched with the corresponding, so far unmatched, parts of the log line. For this purpose, any of the previously described algorithms for generating character-based templates can be used. Lines 3 and 4 in Fig. 7 visualize the final step and line 5 shows the resulting template.

For the evaluation of the algorithm, we chose the merge algorithm, because it provides the most accurate templates among the algorithms, as the evaluation in Sec. 5 shows. The disadvantage of the longer runtime is mitigated, because of the shorter length of the compared strings.

Algorithm 6 describes the implementation of the token_char algorithm. First, the algorithm splits log line S_2 into tokens and transforms it into a token structure T_2. Then it performs the token matching between the current template T_1 and the token-structure of log line T_2. In this step, the algorithm also generates the character structure of the log line. The algorithm compares the character

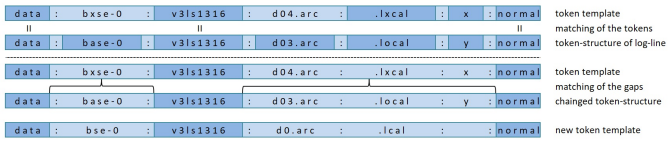


Figure 7: Token_char algorithm matching. Dark blue parts represent token-structures and light blue parts character-structures. Colons represent any fixed set of predefined delimiters.

structures $string\_template_1$ of the current template and their corresponding character-structures $string\_template_2$ of the log line in a for loop. For that purpose, the algorithm iterates over the gaps of the token-structures T_1 and T_2, which as mentioned refer to the character-structures. For matching the character-structures, the algorithm applies Alg. 2. Finally, the resulting alignment of the character-structures replaces the corresponding character-structure $string\_template_1$ in the current template T_1.

Algorithm 6 TOKEN_CHAR(T_1, S_2)

1: $T_2 \leftarrow$ SPLIT_INTO_TOKENS(S_2);
2: TOKEN_MATCHING(T_1, T_2);
3: **for** $(string\_template_1, string\_template_2) \in Gaps(T_1, T_2)$ **do**
4: Replace $string\_template_1$ in T_1 with MERGE($string\_template_1$, $string\_template_2$);
5: **end for**
6: **return** T_1

5 EVALUATION

The following section presents the evaluation of our approach for generating character-based cluster templates. First, we describe the data used for the evaluation. Next, we define a similarity score that we calculate alongside the F-score to asses accuracy and quality of the algorithms introduced in Sec. 4. Finally, we interpret and discuss the evaluation results. All evaluations have been carried out on a Notebook with an Intel Core i7-5600U CPU 2.60 GHz and 16 GB RAM running Windows 7 64-Bit. The assessed algorithms have been implemented in Python 3.7.

5.1 Test data

For the evaluation of our approach, we use three different data sets. This demonstrates the broad applicability of the approach for various log data types. The first data set, we refer to as *DS-A*, originates from a network that runs a MANTIS Bug Tracker System[2]. Therefore, the data set contains logs from an Apache Web server hosting the MANTIS platform, a MySQL database, a reverse proxy and a firewall, as well as a mail server. The log messages of these systems are aggregated using syslog. The data set consists of 1.6 million log lines that reflect 10 hours of system usage. The second data set, we refer to as *DS-B*, derives from the same system. DS-B includes the syscalls of the system, which have been collected

[2]https://www.mantisbt.org/

Table 1: Properties of the subsets of the described data used for evaluation. For the line length, the number of words (space separated substrings) and the cluster size, the table provides values for minimum, average and maximum.

	DS-A	DS-B	DS-C
data set size	10.000	133.000	200.000
line length	60 / 135.94 / 1959	79 / 211.10 / 328	92 / 139.03 / 311
word #	3 / 12.60 / 133	8 / 32.67 / 58	9 / 13.72 / 31
cluster #	352	180	21
cluster size	1 / 28.41 / 605	1 / 741.47 / 13857	1 / 9523.81 / 46585

using the auditd service. The third data set, we refer to as *DS-C*, includes logs from a Hadoop File System running on a 203-node cluster on Amazon's EC2 platform [13]. DS-C consists of 11 million lines that reflect almost 3 days of system behavior. Since the evaluated algorithms require pre-clustered data, we clustered the data applying the incremental clustering approach from [12], using a similarity threshold of 0.9 for DS-A and DS-B, and 0.6 for DS-C. We selected the similarity threshold with respect to the structure and complexity of the data. We chose a lower similarity for DS-C, because the data set includes larger variable parts and a higher similarity threshold would lead to a large number of small clusters that would represent an inappropriate cluster arrangement that includes many similar clusters.

5.2 Evaluation metrics

We used two different evaluation metrics to assess and compare the different algorithms. The first one is a score for similarity, which is defined in the following, and the second one is the F-Score.

The Sim-Score measures the similarity between the log lines of a cluster and its corresponding template. The algorithms for generating character-based templates provide templates that match all log lines of a cluster. Therefore, the ratio between the number of characters the template consists of and the average log line length is a measure for similarity between a template and the log lines of a cluster. In the Sim-Score, the average log line length corresponds to the mean of the number of characters the log lines of a cluster consist of. Consequently, the resulting Sim-Score for each algorithm is the mean of these similarities. The Sim-Score is calculated as shown in Eq. (1), where n is the number of clusters, m_i the number of log lines in the i-th cluster, T_i the template of the i-th cluster, $L_{i,j}$ is the j-th log line of the i-th cluster and $|\cdot|$ denotes the number of characters of a template or a log line.

$$\text{Sim-Score} = \frac{1}{n}\sum_{i=1}^{n}\frac{|T_i|}{\frac{1}{m_i}\sum_{j=1}^{m_i}|L_{i,j}|} \tag{1}$$

The Sim-Score is an evaluation metric that indicates how accurate the templates are. One advantage of the Sim-Score is that it does not rely on any additional information about the clusters, such as a ground truth, which defines the optimal template. Thus, it can be calculated directly after generating templates, for any data set. Table 1 presents properties of the data we used for evaluating the Sim-Score.

The second metric we used to evaluate the proposed algorithms for generating character-based templates is the F-score (see Eq. (2)). The F-Score allows an assessment of the accuracy of the generated templates. However, in opposite to the Sim-Score, the calculation of the F-Score requires a ground-truth to identify true positives (TP), false postives (FP) and false negatives (FN), as Eq. (2) indicates. Therefore, we first had to create a character-based ground truth for all data sets.

$$F\text{-score} = \frac{2TP}{2TP + FN + FP} \tag{2}$$

Furthermore, we defined the terms TP, FP and FN[3]:

- TP are defined as the characters which appeared in the same order in both the ground truth and the created templates.
- FP are characters, which occur in the template but not in the ground truth.
- FN are characters, which occur in the ground truth but not in the template.

FP are an issue that cannot simply be ignored. The major reason for FP are over-fitting templates. The algorithms tend to create overly accurate templates, because they only generate them from the perspective of the log lines that are associated with a cluster and not taking other knowledge into account as humans would do. Reasons for this are characters that actually represent variable parts of a log line, but occur in each log line of a cluster. However, these characters are not part of the ground truth, because they, for example, refer to an IP address or a part of a timestamp, which might only be static for the training data and thus are not considered static in the ground truth. An example is a variable within the same cluster that takes the values `192.67.200.155` and `192.67.200.12`. In this case, `192.67.200.1` becomes part of the template, although the last character 1 belongs to a variable part of a log line. Hence, the resulting template would not match the IP address `192.67.200.2`, which might be also valid.

5.3 Sim-Score evaluation results

The following section discusses the results of the evaluation of the Sim-Score. As previously mentioned, the calculation of the Sim-Score does not require any additional information, such as a predefined ground truth. Thus, the Sim-Score is suitable to be calculated for any log data set. Furthermore, to compare the character-based template generators with a token-based approach, we also generated token-based templates, using the part of the token_char algorithm that generates the token-structure of the template.

Tables 2, 3 and 4 present the evaluation results of the Sim-Score for the different template generator algorithms using the data sets described in Tab. 1. As expected, the proposed character-based algorithms yield a much higher Sim-Score than the token-based approach. However, the token_char provides comparable results to the pure character-based algorithms. The differences between the Sim-Scores of the character-based algorithms are so small that they can be neglected. Nevertheless, the results of the runtime are of greater significance. The merge algorithm shows the longest runtime among all tested algorithms. This is the case, because all other character-based algorithms first divide the line into shorter segments by marking parts of the line that are equal to tokens of the template. Then, they match the remaining shorter parts of the line and the template by calculating the LV-distance. Whereas,

[3]Since gaps can be optional they do not influence the Sim-Score.

Table 2: Sim-Score comparison on DS-A

	merge	length	equalmerge	token_char	token
Sim-Score	96.38%	96.24%	96.37%	95.18%	85.27%
Time (s)	435.20	23.46	25.52	29.54	8.49

Table 3: Sim-Score comparison on DS-B

	merge	length	equalmerge	token_char	token
Sim-Score	91.40%	90.71%	91.42%	91.42%	77.27%
Time (s)	35179.35	55.56	63.37	843.51	366.76

Table 4: Sim-Score comparison on DS-C

	merge	length	equalmerge	token_char	token
Sim-Score	71.96%	70.41%	71.95%	71.96%	52.67%
Time (s)	11207.57	344.21	227.22	1387.87	154.14

the merge algorithm calculates the LV-distance for the whole log line and the whole template. While the length and the equalmerge algorithms showed a comparable runtime on the data sets DS-A and DS-B, the equalmerge algorithm outperforms all the others on DS-C. Due to the lower similarity threshold during clustering and larger variable parts in the log data, Sim-Scores for DS-C decrease for all algorithms. Furthermore, the larger variable parts in DS-C are the reason, why the equalmerge algorithm outperforms the length algorithm. While the equalmerge algorithm merges the blocks that are not marked and then calculates the LV-distance, the length algorithm first has to localize all blocks of the current template in the log line at hand. Due to the large variable parts the number of blocks the template consists of increases in every step. Furthermore, the different sizes of the data sets affect the token_char approach more than the others. The reason for this is, that the token_char algorithm has to do the matching for the token-structure and all character-structures of the template. The runtime of the pure token-based approach is rather long when processing DS-B. This is, because of the long lines consisting of a large number of tokens.

5.4 Scalability

The next section summarizes results on the evaluation of the scalability of the different algorithms. Figure 8 visualizes the results for the different algorithms, showing the number of lines on the x-axis and the runtime on the y-axis. For the evaluation of the scalability, we chose a cluster from DS-B that comprises more than 1000 log lines. Then, we measured the runtime it took to calculate the template for the cluster for 5 to 1000 lines in steps of 50 lines. Figure 8 demonstrates that the runtime of all algorithms scales linearly with respect to the cluster size m, which results in a computational complexity of $O(m)$. Figures 8b and 8c demonstrate that the length and the equalmerge algorithm scale equally well with respect to the runtime and gradient, followed by the token_char algorithm in Fig. 8d. The merge algorithm, see Fig. 8a, has the worst runtime and gradient.

5.5 Cluster arrangement

We also investigated the impact of the order of the log lines in a cluster on the resulting template and the process of generating it.

Table 5: Cluster arrangement

	original	maxfirst	maxdist	mindist
merge Sim-Score	96.38%	96.44%	96.43%	96.47%
length Sim-Score	96.24%	96.41%	96.40%	96.26%
equalmerge Sim-Score	96.37%	96.42%	96.41%	96.44%
token_char Sim-Score	95.18%	96.38%	96.37%	96.38%
merge last change	82.76%	76.20%	71.35%	93.55%
length last change	82.06%	74.96%	70.57%	92.93%
equalmerge last change	82.21%	74.94%	70.66%	93.06%
token_char last change	70.87%	67.84%	67.25%	78.63%

Therefore, we changed the order of the log lines in the clusters as follows:

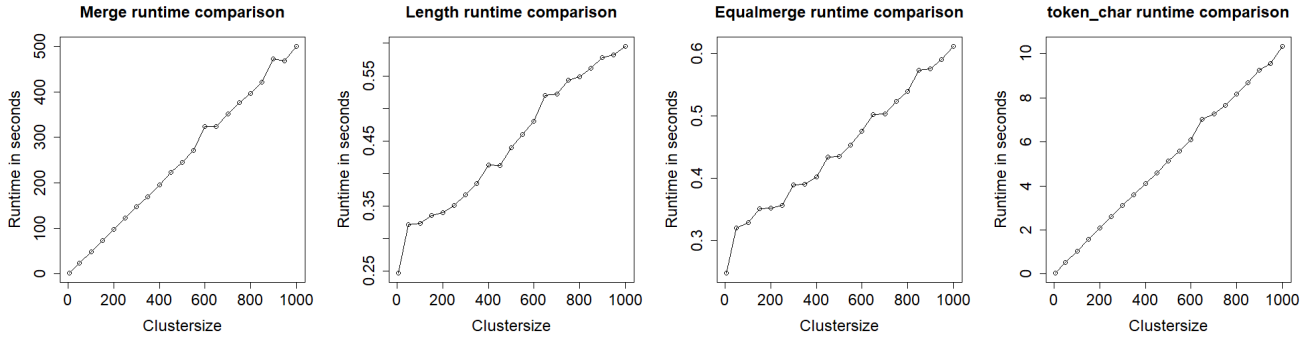
(i) The original order (original),
(ii) starting with the two lines that have the maximum LV-distance in the whole cluster and the following lines have the original order (maxfirst),
(iii) ordering the lines by the LV-distance to each other starting with the line that has the largest distance to the others (maxdist),
(iv) ordering the lines by the LV-distance to each other starting with the line that has the smallest distance to the others (mindist).

Table 5 summarizes the results of the cluster arrangement evaluation carried out on the first 10.000 lines of the data set DS-A. The lower part of the Table shows after processing which percentage of log lines in a cluster the template does not change any more. Our evaluation proves, that the order has impact on the number of processed log lines after which the template does not change anymore and therefore on the runtime. Ordering the lines by the LV-distance to each other starting with the line that has the largest distance to the others (maxdist) showed the best results, closely followed by starting with the two lines that have the maximum LV-distance and the following lines have the original order (maxfirst). Those two approaches improve the runtime in opposite to keeping the original order, while using the mindist approach increases the runtime. However, there was virtually no impact on the Sim-Score as the upper part of Tab. 5 points out. Since ordering the lines within a cluster by their LV-distance is computational expensive with $O(n^n)$, where n is the number of lines, the runtime improvement can only be realized when the lines are already in the correct order.

Additionally, Fig. 9 visualizes the progression of the change in the number of characters the template of a representative cluster consists of. Therefore, we plotted the number of characters the current template exists of over the number of processed lines for the four different cluster arrangements. The figure demonstrates that for the maxfirst and maxdist arrangement the template gets stable after a few lines, while the mindist arrangement, requires major changes in the template towards the end. The original arrangement lies between the others.

5.6 Evaluation of different data set sizes

In this section, we evaluate the influence of the data set size on the resulting templates. Therefore, we compared the Sim-Score of the



(a) Merge algorithm (b) Length algorithm (c) Equalmerge algorithm (d) Token_char algorithm

Figure 8: Runtime comparison.

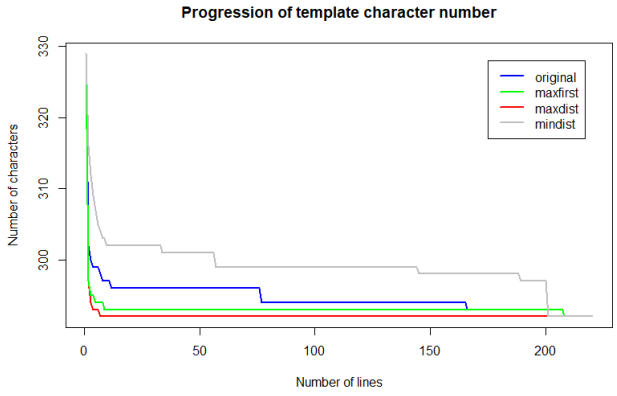


Figure 9: Progression of cluster template character number

Table 6: Evaluation of different datasets.

data size	10K	50K	1600K
merge Sim-Score	96.38%	95.79%	94.99%
length Sim-Score	96.24%	95.44%	94.40%
equalmerge Sim-Score	96.37%	95.71%	94.73%
token char Sim-Score	95.18%	93.53%	93.26%

whole data set DS-A, a subset consisting of the first 10.000 lines and a subset of the first 50.000 lines. The results, summarized in Tab. 6, indicate a small decline in the Sim-Score with increasing data set size. This can be explained as follows: The larger the data set, the more log lines are assigned to each cluster. Therefore, the similarity of the log lines within a cluster decreases, which as described in Sec. 5.4, affects the Sim-Score of the template. But, the lower Sim-Scores do not refer to templates of lower quality. Indeed, while the Sim-Score only slightly decreases, over-fitting is reduced. Hence, the quality of the templates actually increases, because of the more diverse set of log lines, which more accurately reflects the system behavior. Finally, we can conclude that the data set size does not strongly affect the quality of the resulting template.

5.7 Robustness

Furthermore, we evaluated the robustness of the algorithms, which is especially important for the length and the equalmerge algorithm. Since these two algorithms first mark parts of the template that equally occur in the currently processed log line, they imitate longest common subsequence [1]. This might cause problems, if the lines within a cluster are different, but substrings in different positions are marked as equal, due to the fact that there are many variable parts in the log lines. Considering the strings `ayyaa`, `aayya`, `aaa`, the optimal template would be `a[*]a[*]a`, but because the first created template would be `[*]ayya[*]`, the final template becomes `[*]a[*]a[*]`, which leads to a lower similarity between the strings and the template.

Additionally, the localizing step in the length and equalmerge algorithm could be erratic, when the template includes two equal blocks, that only occur once in the currently processed log line. Therefore, a false marking can happen. For example, considering the strings `stringstring`, `string string` and `tring string`. The first two yield the template `string[*]string`, but because the algorithm localizes the first block in the rearmost part of the log line, the second block is marked with the empty string. Thus, the created template would be `[*]string[*]`, although `[*]tring[*]string` would be the the optimal one.

Hence, we ran the algorithms on the first 10k lines of data set DS-A, which was clustered using different similarity thresholds. The lower the threshold during clustering, the more dissimilar are the log lines within a cluster. In this way we can evaluate the effects of the marking step in the length and equalmerge algorithm, because which blocks are marked as substrings in the log line depends on the similarity of the log lines in a cluster. The effects can be seen when comparing the Sim-Score of the length and the equalmerge algorithm with the results of the merge algorithm, which does not include the marking step.

Table 7 demonstrates that there is no extensive decrease of the Sim-Score in either of the algorithms, which is only the case if the marking had a severe impact. Therefore, all of the algorithms can be considered robust.

Table 7: Robustness evaluation for different minimum similarities between log lines within a cluster.

similarity	0.9	0.8	0.7	0.6	0.5
merge	96.38%	91.14%	82.76%	74.97%	71.09%
length	96.24%	90.48%	81.25%	73.76%	68.49%
equalmerge	96.37%	90.98%	82.34%	74.31%	70.64%
token_char	95.18%	90.47%	82.04%	73.85%	69.41%

Table 8: Test against character-based ground truth: H F-score is the F-score for the Hadoop data set and TB F-score the F-score for the thunderbird data set.

	merge	length	equalmerge	token_char	token GT
H F-score	0.9910	0.9902	0.9910	0.9910	0.8853
TB F-score	0.9958	0.9941	0.9958	0.9958	0.9296

5.8 F-score evaluation

Since the F-score requires a ground truth, we chose data sets from Hadoop and Thunderbird available on the Internet [7], where these data sets each have 2000 lines and the corresponding token-based ground truths are also available. We created the character-base ground truths, which are the optimal template, based on the token-based ground truths.

The F-score was calculated for each algorithm as described in Sec. 5.2. Furthermore, we also tested the token-based ground truth against the character-based one. Since the token-based ground truth (token GT) is the optimum which token-based template generators can achieve, the resulting F-Score is the maximum any token-based approach can reach.

Table 8 presents the results of the F-score evaluation. The evaluation proves that all character-based algorithms yield more accurate templates than a token-based ever could. Merge, equalmerge and token_char provide the best F-score, followed by the length algorithm and the token ground truth. The F-scores of merge and equalmerge are the same, because they both created the same templates for these sets of log data. The token_char algorithm also had the same F-score, but yielded different templates, because it placed the gaps differently.

5.9 Feature Analysis

Finally, we assess the features of the different algorithms with respect to performance and accuracy, which are summarized in Tab. 9. The merge algorithm provides the most accurate template according to *Sim*-Score and F-score. However, it lacks performance and therefore should not be applied for time critical tasks. The length algorithm provides comparable accurate templates, while optimizing performance in opposite to the merge algorithm. The performance boost is achieved by marking blocks of the current template that occur as substrings in the log line. Therefore, the length of the strings for which the LV-distance has to be calculated, can be significantly reduced. The equalmerge algorithm combines the length and the merge algorithm and performs almost as good as the length algorithm, while providing templates that are almost as accurate as the ones computed by the merge algorithm. The token_char algorithm performs slightly better than the merge algorithm, but is surpassed by the performance of the length and equalmerge algorithms. Moreover, the templates provided by the pure character-based approach are more accurate. Hence, we recommend for any application to apply the equalmerge algorithm instead of the token_char approach. The pure token-based approach shows the best performance, while providing the least accurate templates. Additionally, all the disadvantages mentioned in the Sec. 1 have to be considered when applying token-based approaches for generating templates.

Table 9: Comparison of performance and accuracy.

Algorithm	Performance	Accuracy
merge	- -	++
length	+	+
equalmerge	+	++
token_char	∼	+
token	++	- -

6 CONCLUSION AND FUTURE WORK

In this paper we introduced a novel approach for generating character-based templates for computer log data. The goal was to provide meaningful cluster descriptions that support further manual and automatic analysis of clustered log data, such as review of the current system behavior by a system administrator and parser generation to enable, for example, anomaly detection. Hence, to achieve this goal, we had to develop a method to calculate multi-line alignments for any group of strings. For this purpose, we designed four different algorithms that combine comparison-based procedures with heuristics to compute approximations of the optimal multi-line alignments.

In a detailed evaluation carried out on three different log data sets, we calculated a newly defined Sim-Score and the F-Score for the four different approaches. The results show the high quality of the character-based templates. All algorithms reached an F-Score higher than 0.99. Furthermore, we demonstrated linear scalability with respect to the number of lines within a cluster and showed the robustness of our algorithms. We also analyzed the influence of the length of the data sets and the processing order of the log lines. Finally, we conclude that the equalmerge approach yielded the best results regarding performance and accuracy.

In the future, we plan to apply the proposed approach for generating character-based templates for log clusters, with purpose of improving the generation of log data parsers and to enhance time series analysis carried out on log data.

ACKNOWLEDGMENTS

This work was partly funded the EU H2020 project GUARD (833456) and by the FFG projects DECEPT (873980) and INDICAETING (868306).

REFERENCES

[1] Wael H Gomaa and Aly A Fahmy. 2013. A survey of text similarity approaches. *International Journal of Computer Applications* 68, 13 (2013), 13–18.

[2] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. 2016. An Evaluation Study on Log Parsing and Its Use in Log Mining. In *DSN'16: Proc. of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[3] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 33–40.

[4] D. Jurafsky and J.H. Martin. 2009. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall.

[5] Max Landauer, Florian Skopik, Markus Wurzenberger, and Andreas Rauber. 2020. System log clustering approaches for cyber security applications: A survey. *Computers & Security* 92 (2020), 101739. https://doi.org/10.1016/j.cose.2020.101739

[6] Max Landauer, Markus Wurzenberger, Florian Skopik, Giuseppe Settanni, and Peter Filzmoser. 2018. Dynamic log file analysis: an unsupervised cluster evolution approach for anomaly detection. *computers & security* 79 (2018), 94–116.

[7] LogPAI. 2019. *Log Analytics Powered by AI*. Retrieved June 26, 2019 from https://github.com/logpai/logparser

[8] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2009. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1255–1264.

[9] Cédric Notredame. 2007. Recent evolutions of multiple sequence alignment algorithms. *PLoS computational biology* 3, 8 (2007), e123.

[10] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. IEEE, 119–126.

[11] Markus Wurzenberger, Max Landauer, Florian Skopik, and Wolfgang Kastner. 2019. AECID-PG: A Tree-Based Log Parser Generator To Enable Log Analysis. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 7–12.

[12] Markus Wurzenberger, Florian Skopik, Max Landauer, Philipp Greitbauer, Roman Fiedler, and Wolfgang Kastner. 2017. Incremental clustering for semi-supervised anomaly detection applied on log data. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 31.

[13] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jor dan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 117–132.