Towards String Sanitization

Oluwole Ajala, Hayam Alamro, Costas Iliopoulos, Grigorios Loukides

▶ To cite this version:

Oluwole Ajala, Hayam Alamro, Costas Iliopoulos, Grigorios Loukides. Towards String Sanitization. 14th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI), May 2018, Rhodes, Greece. pp.200-210, 10.1007/978-3-319-92016-0_19 . hal-01821302

HAL Id: hal-01821302

https://inria.hal.science/hal-01821302v1

Submitted on 22 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Towards String Sanitization

Oluwole Ajala, Hayam Alamro, Costas Iliopoulos, and Grigorios Loukides

Department of Informatics, Kings College London, London, UK,
[oluwole.ajala, hayam.alamro, costas.iliopoulos,
grigorios.loukides]@kcl.ac.uk

Abstract. An increasing number of applications, in domains ranging from biomedicine to business and to pervasive computing, feature data represented as a long sequence of symbols (string). Sharing these data, however, may lead to the disclosure of sensitive patterns which are represented as substrings and model confidential information. Such patterns may model, for example, confidential medical knowledge, business secrets, or signatures of activity patterns that may risk the privacy of smart-phone users. In this paper, we study the novel problem of concealing a given set of sensitive patterns from a string. Our approach is based on injecting a minimal level of uncertainty to the string, by replacing selected symbols in the string with a symbol "$*$" that is interpreted as any symbol from the set of possible symbols that may appear in the string. To realize our approach, we propose an algorithm that efficiently detects occurrences of the sensitive patterns in the string and then sanitizes these sensitive patterns. We also present a preliminary set of experiments to demonstrate the effectiveness and efficiency of our algorithm.

Keywords: string, sanitization, data privacy

1 Introduction

Data that are used in a number of applications such as mining and web analysis often come in the form a long sequence of events. In many of these applications, such as biomedical informatics, network analysis, and marketing, the patterns of interest in analysis come in the form of consecutive symbols, i.e., substrings of the long sequence (string). At the same time, the data also contains patterns (substrings) that represent confidential information that must be protected before the data are published (released) for analysis. Examples of such information are medical knowledge (diagnosis and medications) and business secrets.

In this work, we study the general problem of sanitizing a string. The problem deviates from existing works, which focus on the sanitization of event sequences, or relational tables (i.e., collections of records). We formalize the problem of protecting a string by *sanitizing* (i.e., concealing) a given set of sensitive patterns that represent confidential information. To achieve this, we replace selected symbols in all occurrences of the sensitive patterns in the string with a symbol "$*$". That is, we inject uncertainty in the string. This clearly reduces the usefulness of the protected string in applications.

Thus, in our problem, we aim to conceal the sensitive patterns while preserving the utility of the string as much as possible. As we show, the problem is NP-hard. To address the problem, we propose an algorithm, called SSA (String Sanitization Algorithm) that first efficiently detects occurrences of the sensitive patterns in the string and then sanitizes these sensitive patterns. The detection of the sensitive patterns is performed by the Aho-Corasick algorithm [3], which takes linear time in the length of the string. The sanitization of the sensitive patterns is performed by a greedy algorithm that is inspired by the well-known greedy algorithm for the Set Cover (SC) problem [5]. This algorithm is called SanitizeClusters, and it works by iteratively replacing the symbol that is contained in the largest number of currently unprotected sensitive patterns with the special symbol "$*$".

We also present a preliminary set of experiments to demonstrate the effectiveness and efficiency of our algorithm. In our experiments, we apply the algorithm to a commonly used sequential dataset and study empirically its effectiveness in terms of preserving data utility, as well as its efficiency. Our results show that the algorithm can sanitize the dataset in less than 10 milliseconds and produce a sanitized dataset with a small number of occurrences of the symbol "$*$".

The organization of the rest of the paper is as follows. Section 2 summarizes the related work and state of art of this current research. In Section 3, we present our preliminaries containing formal definitions and an overview of the problem statement. We highlight our approach in Section 4, where we present our algorithms, their implementation and examples. We present a preliminary experimental evaluation of our approach in Section 5. Finally, we conclude and discuss future work in Section 6.

2 Related Work

Data sanitization is an important methodology to protect data, by means of concealing confidential knowledge in the form of user-specified patterns. Most existing sanitization approaches are applied to a collection (multi-set) of transactions [15, 19, 20], sequences [2, 8, 11], or trajectories [2] and prevent the mining of frequent sensitive patterns [2, 8, 19] or association rules [15, 20], by applying deletion. To preserve utility, these approaches attempt to minimize the total number of deleted items (events) [8] and/or changes in the frequency of nonsensitive patterns [2], as well as in the output of frequent pattern [8, 11, 19] or association rule [15, 20] mining. We share the utility goal of [8], but contrary to all these approaches, we consider a long string of events.
This is somewhat similar to the work of [13], which considered a sequence of events that are associated with time points. However, our work differs from that of [13] along two important dimensions. The first dimension is related to the type of data considered. Specificailly, we consider a string (sequence of symbols), whereas the work of [13] considers a sequence of multisets of symbols, where each multiset is associated with a time point. The second dimension is related to the problem considered. Specifically, the problem we aim to solve requires concealing sensitive strings (i.e., sequences of consecutive symbols that can potentially contain gaps), while minimizing the total number of deleted events. On the ontrary, the problem of [13] requires concealing single events (symbols), while preserving the distribution of event in the entire sequence. Also, we

consider a sensitive string concealed when it does not appear in the entire string (i.e., has frequency zero), whereas the work of [13] considers a sensitive event concealed when its frequency in the event sequence is below a given threshold in any prefix of the sequence. The two problems also belong to different complexity classes; our problem is strongly NP-hard, as we show, and thus cannot be solved optimally in polynomial time, whereas the problem in [13] is weakly NP-hard and thus admits a pseudopolynomial time optimal algorithm.
Anonymization is a different methodology to protect data, whose goal is not to conceal given sensitive patterns but to preserve the privacy of individuals, whose information is contained in the data, by preventing inferences of the identity and/or sensitive information of these individuals. Most anonymization approaches are applicable to a collection of transactions [4, 9], trajectories [1, 16], or sequences (with [18] or without [14] time points), each of which is associated with a different individual. Another category of approaches anonymizes an individual's time-series [17] or event sequence [10], using differential privacy [7]. Anonymization approaches guard against the disclosure of an individual's identity [1, 14] and/or sensitive information [1, 4, 10, 17, 18]. However, they cannot be applied to our problem, because their privacy models do not conceal the sensitive patterns in a long string, which we require to preserve privacy. Suppressing sensitive patterns from an infinite event sequence has been studied in [12, 21]. These works aim to achieve privacy by minimizing the number of occurrences of sensitive patterns, while preserving the occurrences of certain nonsensitive patterns that are specified by data owners. Both types of patterns are sets of events. Thus, these works are not applicable to our problem, because they consider different type of data and have different utility goals.

3 Background and problem definition

3.1 Background

We begin with basic definitions and notation from [6]. Let $x = x[0]x[1] \dots x[n-1]$ be a *string* of length $|x| = n$ over a finite ordered alphabet. We consider the case of strings over an *integer alphabet*: each letter is replaced by its lexicographical rank in such a way that the resulting string consists of integers in the range $\{1, \dots, n\}$. For two positions i and j on x, we denote by $x[i \dots j] = x[i] \dots x[j]$ the *factor* (sometimes called *substring*) of x that starts at position i and ends at position j (it is of length 0 if $j < i$), and by ε the *empty string* of length 0. We recall that a *prefix* of x is a factor that starts at position 0 ($x[0 \dots j]$) and a *suffix* of x is a factor that ends at position $n-1$ ($x[i \dots n-1]$). A prefix (resp.suffix) is said to be *proper* if it is any prefix (resp. suffix) of the string other than the string itself. For example, any of the strings in {a,ac, acc, accg} (respectively, {ccgt, cgt, gt, t}) is a proper prefix (respectively, suffix) of the string $accgt$.

We denote with S a string that we aim to protect and with $\mathcal{S}$ the set of symbols contained in S. To protect the string S, we conceal a set of substrings, $SP = \{sp_1, \dots, sp_n\}$, of S. We refer to the substrings of SP as sensitive patterns, and we denote the set of symbols $\cup_{i \in [1,n]} sp_i$ with $\mathcal{SP}$. To conceal a sensitive pattern sp_i, we replace at least one symbol in it with a symbol $* \neq \varepsilon$. This process constructs a *sanitized*

sensitive pattern sp'_i corresponding to sp_i. If sp_i occurs multiple times in the string S, we replace each occurrence of sp_i with sp'_i. Doing this for every sensitive pattern in SP constructs a *sanitized* string S' corresponding to S. Clearly, we cannot be certain about the symbols that were replaced by the symbol " $*$" in the sanitized string S'. This helps preserving privacy, because a recipient of S' interprets " $*$ " as any symbol of $\mathcal{S}$ (assuming that $\mathcal{S}$ is public knowledge and contains more than one symbol).

3.2 Problem definition

In this section, we formally define the probem we study and show that it is NP-hard.

Problem 1 (Optimal String Sanitization (OSS)). Given a string S and a set of sensitive patterns $SP = \{sp_1, \dots, sp_n\}$, construct a *sanitized* string S' from S, such that: (I) S' contains no $sp_i \in SP$, $i \in [1, n]$, as a substring, (II) S' contains each sanitized pattern, sp'_i, $i \in [1, n]$, that is constructed from sp_i by replacing at least one symbol in sp_i with the symbol $*$, as a substring, and (III) the number of occurrences (multiplicity) of the symbol $*$ in S' is minimum.

Theorem 1. *The OSS problem is NP-hard.*

Proof. The proof is by reducing the NP-hard Set Cover (SC) problem to OSS. The SC problem is defined as follows. Given a universe of elements $U = \{u_1, \dots, u_n\}$ and a collection $L = \{L_1, \dots, L_m\}$ such that each $L_j \in L$ is a subset of U, find a subcollection $L' \subseteq L$, such that: (I) L' covers all elements of U (i.e., $\cup_{L_j \in L'} L_j = U$), and (II) the number of subsets in L' is minimum.

We map a given instance $\mathcal{I}_{SC}$ of SC to an instance $\mathcal{I}_{OSS}$ of the OSS problem, in polynomial time, as follows:

I Each element $u_i \in U$ is mapped to a sensitive pattern $sp_i \in SP$, so that covering the element u_i corresponds to constructing a santized pattern sp'_i from sp_i.

II Each subset $L_j = \{u_1, \dots, u_r\} \in L$ is mapped to a set $SP_j = \{sp_1, \dots, sp_r\}$ $\subseteq SP$ of sensitive patterns which have a common symbol $s_j \in \mathcal{S}$, so that selecting L_j corresponds to constructing a set of sanitized patterns $\{sp'_1, \dots, sp'_r\}$ by replacing the common symbol s_j of the patterns in SP_j with the symbol $*$.

In the following, we prove the correspondence between a solution L' to the given instance $\mathcal{I}_{SC}$ of SC and a solution S' to the instance $\mathcal{I}_{OSS}$.

We first prove that, if L' is a solution to $\mathcal{I}_{SC}$, then S' is a solution to $\mathcal{I}_{OSS}$. Since $\cup_{L_j \in L'} = U = \{u_1, \dots, u_n\}$, a sanitized sensitive pattern sp'_i is constructed for each sensitive pattern sp_i in SP. Thus, S' contains no sensitive pattern in SP as a substring, and it contains each sanitized sensitive pattern sp'_i that corresponds to a sensitive pattern sp_i, $i \in [1, n]$, as a substring. Since the number of subsets in L' is minimum and each $L_j \in L'$ leads to the replacement of the common symbol s_j of the patterns in SP_j with $*$, the sensitive patterns in SP are sanitized with the minimum number of occurrences of the symbol $*$. Furthermore, all occurences of a sensitive pattern sp_i must be replaced with its corresponding sensitive pattern sp'_i. Thus, the number of occurrences of the symbol $*$ contained in the sanitized string S' is minimum, and S' is a solution to $\mathcal{I}_{OSS}$.

We now prove that, if S' is a solution to $\mathcal{I}_{OSS}$, then L' is a solution to $\mathcal{I}_{SC}$. Since S' is a solution to $\mathcal{I}_{OSS}$, each sensitive pattern in SP is sanitized, which implies that L' covers all elements of U. In addition, the number of occurrences of the symbol $*$ in S' is minimum. This implies that the sensitive patterns in SP are sanitized with the minimum number of occurrences of the symbol $*$ and hence the number of selected subsets in L' is minimum. Thus, L' is a solution to $\mathcal{I}_{SC}$. □

4 String Sanitization Algorithm

This section discusses the String Sanitization Algorithm (SSA), which aims to solve the OSS problem. The algorithm is based on: (i) the *Aho-Corasick* string matching algorithm whose objective is to efficiently detect the occurrences of all sensitive patterns in the string, (ii) the SetIntersection algorithm whose objective is to organize sensitive patterns into clusters based on the symbols they share, and (iii) the SanitizeClusters algorithm whose objective is to perform the sanitization of the sensitive patterns in each cluster by replacing selected symbols in the patterns by the symbol "$*$".

4.1 Aho-Corasick algorithm

The Aho-Corasick algorithm [3] is a well-known, efficient algorithm for detecting all occurrences of a finite set of patterns $P = \{p_1, \ldots, p_k\}$ in a given text T. Both patterns and T are strings. The algorithm constructs a finite state machine (FSM) automaton for the set of patterns P, which is used to perform pattern matching on the text T (i.e., find the symbol of T at which each occurrence of each pattern in P occurs in T). The benefit of the Aho-Corasick algorithm is that it works in linear time (when the string is independent of the number of patterns, as in our case). Specifically, the worst-case time complexity of the algorithm is $O(|T| + \sum_{i\in[1,k]} |p_i| + \mathcal{M}_{P,T})$, where $|T|$ denotes the length of T, $\sum_{i\in[1,k]} |p_i|$ denotes the the total length of the patterns in P, and $\mathcal{M}_{P,T}$ denotes the number of occurrences of patterns in P in the text T. This is typically much faster than the naive solution of detecting the occurrences of each pattern in P independnetly, which takes $O(\sum_{i\in[1,k]} |p_i| \cdot |T|)$ time.

In this work, we use the Aho-Corasick algorithm to detect all occurrences of sensitive patterns $\{sp_1, \ldots, sp_n\}$ in the string S, which is needed to before sanitizing the sensitive patterns (i.e., replacing at least one symbol in each sensitive pattern with $*$, in all occurrences of the sensitive pattern in the string). Thus, the worst-case time complexity of the Aho-Corasick algorithm in our case is $O(|S| + \sum_{i\in[1,m]} |sp_i| + \mathcal{M}_{SP,S})$, where $|s|$ is the length of the string S, $\sum_{i\in[1,m]} |sp_i|$ is the total length of the sensitive patterns in SP, and $\mathcal{M}_{SP,S}$ is the number of occurrenses of sensitive patterns in SP in S. Since the sensitive patterns have generally a small number of occurrences (since they model confidential knowledge), the Aho-Corasick algorithm is a good choice for detecting the occurrences of sensitive patterns in the OSS problem.

4.2 SetsIntersection algorithm

After applying the *Aho-Corasick* algorithm to the string S, all occurrences of each sensitive pattern $sp_i \in SP$ are contained in $Occ(SP)$. We denote the j-th occurrence of

sensitive pattern sp_i in the string S with $Occ(sp_i, j)$. If the sensitive pattern sp_i is a substring $sp_i[l \ldots l']$ of S, we denote its occurence with the positions l and l'. Given $Occ(SP)$ and SP, the SetsIntersection algorithm begins by creating an empty two-dimensional array A. Then, it fills A by iterating over each sensitive pattern sp_i and creating a cluster C that contains sp_i together with all other sensitive patterns that share a position with sp_i. The set of sensitive patterns in C is added into the first element of A. The second element of A contains either the first position of each occurrence of a sensitive pattern, if the cluster contains only one sensitive pattern (i.e., no other sensitive pattern shares symbols with the sensitive pattern in the cluster), or the common positions of all occurrences of the sensitive patterns in C otherwise. After considering all sensitive patterns, the algorithm returns the array A.

Algorithm 1 SetsIntersection

1: **Input:** sensitive patterns SP, $Occ(SP)$ containing all occurrences of each sensitive pattern in the string S.
2: **Output:** 2D array A. The i-th record of A contains two elements: the first is the cluster i (set of sensitive patterns), and the second is the positions of S that these sensitive patterns share.
3: Create empty 2D array A
4: **for each** sensitive pattern $sp_i \in SP$
5: create new cluster C_j that contains sp_i
6: Add into C_j all sensitive patterns that share a position with sp_i
7: $A[j][0] \leftarrow C_j$
8: **if** C_j contains one sensitive pattern **then**
9: $A[j][1] \leftarrow$ first position of each occurrence of the sensitive pattern
10: **else**
11: $A[j][1] \leftarrow$ all common positions of each occurrence of the sensitive patterns in C_j and the first position of each occurrence of a sensitive pattern in C_j
12: **end if**
13: **return** A

4.3 SanitizeClusters algorithm

This algorithm gets as input the array A created by the SetsItersection algorithm, together with the string S. First, it initializes the sanitized string S' with the original string S. Then, in the for loop, it iteratives over each cluster (set of sensitive patterns contained in the first element of each record in the array A), and it sanitizes the sensitive patterns in the cluster. The sanitization of the sensitive patterns in the cluster (see do while loop) is performed until every sensitive pattern in the cluster has at least one symbol replaced by $*$. To sanitize the sensitive patterns in the cluster, the algorithm finds the most frequent position p corresponding to the sensitive patterns in the cluster and replaces the position with $*$ in S'. If there are more than one most frequent positions, the algorithm selects the last position, for efficiency. After all clusters are sanitized, the algorithm returns the sanitized string S'.

Algorithm 2 SanitizeClusters

1: **Input:** Array A created by Algorithm 1, string S
2: **Output:** Sanitized string S'
3: $S' \leftarrow S$
4: **for each** cluster C contained in the first column of A
5: **do**
6: $p \leftarrow$ most frequent position in C (break ties with last most frequent position)
7: Replace p with $*$ in S'
8: **while** a sensitive pattern in C does not have at least one symbol replaced with * in S'
9: **return** S'

4.4 SSA algorithm

We are now ready to present the SSA algorithm. The algorithm gets as input the string S and the set of specified sensitive patterns SP, and it begins by calling the Aho-Corasick algorithm, to obtain a set of positions $Occ(SP)$, for each sensitive pattern in SP. Then, SSA calls the SetsIntersection algorithm using $Occ(SP)$ and SP, to create the array A that organizes the sensitive patterns into clusters that are associated with their corresponding positions. Next, the array A together with the string S is given as input to the SanitizeClusters algorithm, which produces the sanitized string S'. Last, the SSA algorithm returns S'.

Algorithm 3 SSA (String Sanitization Algorithm)

1: **Input:** String S, set of sensitive patterns SP
2: **Output:** Sanitized string S'
3: $Occ(SP) \leftarrow$Aho-Corasick(S, SP)
4: $A \leftarrow$SetsIntersection($SP, Occ(SP)$)
5: $S' \leftarrow$SanitizeClusters(A, S)
6: **return** S'

4.5 Example of applying SSA

Suppose we have the string $S = \{aatccagcaactagaattgcaagcctcaaaact\}$, and the set of sensitive patterns $SP = \{sp_1, \ldots, sp_5\} = \{ag, caa, aac, aact, aag\}$, as shown in Fig. 1.

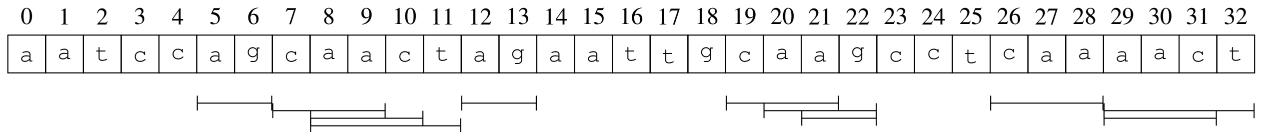

Fig. 1. Occurrences of sensitive patterns $sp_1, \ldots, sp_5$ in the string S.

The SSA algorithm will first execute the *Aho-Corasick algorithm* to obtain $Occ(SP)$, as shown in Fig. 2. After that, the SetsIntersection will be executed to create clusters of sensitive patterns and find the common positions of all occurrences of the sensitive patterns in each cluster (i.e., positions in S that are shared by all patterns in the cluster), if the cluster contains more than one sensitive patterns, or the first position of all occurrences of the sensitive pattern, otherwise. The 5 clusters, together with their corresponding positions in the string S are returned as an array A, as illustrated in Fig. 3.

Occurences	start position	end position
$Occ(sp_1, 1)$	5	6
$Occ(sp_2, 1)$	7	9
$Occ(sp_3, 1)$	8	10
$Occ(sp_4, 1)$	8	11
$Occ(sp_1, 2)$	12	13
$Occ(sp_2, 2)$	19	21
$Occ(sp_5, 1)$	20	22
$Occ(sp_2, 3)$	26	28
$Occ(sp_3, 2)$	29	31
$Occ(sp_4, 2)$	29	32

Fig. 2. $Occ(SP)$ constructed by output of the Aho-Corasick algorithm. $Occ(sp_i, j)$ refers to the j-th occurrence of the sensitive pattern sp_i in the string S and corresponds to a a pair of positions denoting the positions at which the occurrence of sp_i starts and ends in S.

After that, the SanitizeClusters algorithm will consider each cluster and replace the most frequenct of its corresponding positions in the string S with $*$, until all occurrences of all sensitive patterns in the cluster are sanitized (i.e., they have at least one of their positions replaced with $*$). The positionts that will be replaed with $*$ are shown in bold in Fig. 3. Then, the SanitizeClusters algorithm will return the sanitized string S', which is produced by replacing the selected positions with $*$. The sanitized string S' is shown in Fig. 4. Last, S' is output by the SSA algorithm.

cluster	positions
$\{sp_1\}$	$\mathbf{5}, \mathbf{12}$
$\{sp_2, sp_3, sp_4\}$	$8, \mathbf{9}$
$\{sp_2, sp_5, sp_1\}$	$20, \mathbf{21}$
sp_2	$\mathbf{26}$
sp_3, sp_4	$29, 30, \mathbf{31}$

Fig. 3. Array A constructed by the SetIntersection algorithm. The position deleted by the SanitizeClusters appears in bold.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
a	a	t	c	c	*	g	c	a	*	c	t	*	g	a	a	t	t	g	c	a	*	g	c	c	t	*	a	a	a	a	*	t

Fig. 4. Sanitizing sensitive patterns in a given text string after applying SSA

5 Experimental Evaluation

In this section, we evaluate SSA in terms of efficiency (running time) and utility (total number of deleted symbols). We did not compare against existing sanitization methods, as they cannot address the OSS problem. The dataset used was the Activities of Daily Living (ADL) dataset. This dataset is available from the UCI repository (https://archive.ics.uci.edu/ml/), and it has beeen used in prior works on data sanitization [13]. Based on this dataset, the following results were obtained:

Num. of sens. patterns	Total length of sens. patterns	Num. of $*$s	Time (in ms)
5	9	694	5.25
10	17	694	5.75
15	33	447	5.87

Fig. 5. Results for dataset for activity brushing of teeth.

Num. of sens. patterns	Total length of sens. patterns	Num. of $*$s	Time (in ms)
5	11	134	5.60
10	17	219	5.75
15	33	220	5.82

Fig. 6. Results for dataset for activity drinking glass.

Num. of sens. patterns	Total length of sens. patterns	Num. of $*$s	Time (in ms)
5	9	607	5.5
10	17	682	5.77
15	33	895	5.82

Fig. 7. Results for dataset for activity climbing stairs.

As can be seen from Figures 5, 6, and 7, the algorithm is reasonably efficient, requiring no more than 10 miliseconds to sanitize the set of sensitive patterns, which contains no more than 15 patterns. Furthermore, as expected, the runtime generally increases

with the number of sensitive patterns, because, the more sensitive patterns we have, the larger their sum of lengths are (shown as total length of sensitive patterns in the tables).

In addition, the algorithm does not replace a large number of symbols with $*$. Specifically, the number of symbols that were replaced was no more than 895. As expected, the number of symbols that were replaced generally increases with the number of specified sensitive patterns. For example, it increases from 607 to 895 in Figure 7. However, this was not true in Figure 5, because in this experiment the sensitive patterns that were added as the number of sensitive patterns increased from 5 to 15 did not have many of their symbols selected for replacement by the algorithm.

6 Conclusion and Future Work

String sanitization is a necessary task in applications that feature sequences containing confidential information. This paper studied the problem of how to efficiently sanitize a string by replacing a small number of selected symbols contained in sensitive patterns with a special character "$*$". To deal with the problem, we proposed an algorithm SSA that fuses two sub-algorithms, one for detecting occurrences of sensitive patterns in the string, and another for sanitizing the sensitive patterns. The proposed algorithm was implemented and evaluated using the ADL dataset. Our results demonstrate the efficiency and effectiveness of our approach. Being the first work on addressing the problem of sanitizing a string, this work opens up a number of interesting avenues for future investigation. These include: (i) examining how to preserve the utility of the sanitized string in analytics or mining applications, (ii) developing algorithms for sanitizing strings that do not fit into the main memory, and (iii) performing an evaluation of the algorithm using sensitive patterns that are specified by experts and model their privacy requirements (e.g., as in [9, 16]).

References

1. Abul, O., , Bonchi, F., Nanni, M.: Never walk alone: Uncertainty for anonymity in moving objects databases. In: ICDE. pp. 376–385 (2008)
2. Abul, O., Bonchi, F., Giannotti, F.: Hiding sequential and spatiotemporal patterns. TKDE 22(12), 1709–1723 (2010)
3. Aho, A., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM 18(6) (1975)
4. Chen, R., Mohammed, N., Fung, B.C.M., Desai, B.C., Xiong, L.: Publishing set-valued data via differential privacy. PVLDB 4(11), 1087–1098 (2011)
5. Cormode, G., Karloff, H., Wirth, A.: Set cover algorithms for very large datasets. In: CIKM. pp. 479–488 (2010)
6. Crochemore, M., Hancart, C., Lecroc, T.: Algorithm on strings. Cambridge University Press (2007)
7. Dwork, C.: Differential privacy. In: ICALP. pp. 1–12 (2006)
8. Gkoulalas-Divanis, A., Loukides, G.: Revisiting sequential pattern hiding to enhance utility. In: KDD. pp. 1316–1324 (2011)
9. Gkoulalas-Divanis, A., Loukides, G.: Utility-guided clustering-based transaction data anonymization. Trans. Data Privacy 5(1), 223–251 (2012)

10. Götz, M., Nath, S., Gehrke, J.: Maskit: Privately releasing user context streams for personalized mobile applications. In: SIGMOD. pp. 289–300 (2012)
11. Gwadera, R., Gkoulalas-Divanis, A., Loukides, G.: Permutation-based sequential pattern hiding. In: ICDM. pp. 241–250 (2013)
12. He, Y., Barman, S., Wang, D., Naughton, J.F.: On the complexity of privacy-preserving complex event processing. In: PODS. pp. 165–174 (2011)
13. Loukides, G., Gwadera, R.: Optimal event sequence sanitization. In: Proceedings of the 2015 SIAM International Conference on Data Mining. pp. 775–783
14. Monreale, A., Pedreschi, D., Pensa, R.G., Pinelli, F.: Anonymity preserving sequential pattern mining. Artif. Intell. Law 22(2), 141–173 (2014)
15. Oliveira, S.R.M., Zaïane, O.R.: Protecting sensitive knowledge by data sanitization. In: ICDM. pp. 211–218 (2003)
16. Poulis, G., Skiadopoulos, S., Loukides, G., Gkoulalas-Divanis, A.: Apriori-based algorithms for km-anonymizing trajectory data. Trans. Data Privacy 7(2), 165–194 (2014)
17. Rastogi, V., Nath, S.: Differentially private aggregation of distributed time-series with transformation and encryption. In: SIGMOD. pp. 735–746 (2010)
18. Sherkat, R., Li, J., Mamoulis, N.: Efficient time-stamped event sequence anonymization. ACM Trans. Web 8(1), 4:1–4:53 (2013)
19. Sun, X., Yu, P.: A border-based approach for hiding sensitive frequent itemsets. In: ICDM. pp. 426–433 (2005)
20. Verykios, V.S., Emagarmid, A.K., Bertino, E., Saygin, Y., Dasseni, E.: Association rule hiding. TKDE 16(4), 434–447 (2004)
21. Wang, D., He, Y., Rundensteiner, E., Naughton, J.F.: Utility-maximizing event stream suppression. In: SIGMOD. pp. 589–600 (2013)