String Sanitization: A Combinatorial Approach

Giulia Bernardini[1], Huiping Chen[2], Alessio Conte[3], Roberto Grossi[3,4],
Grigorios Loukides[2], Nadia Pisanti[3,4], Solon P. Pissis[4,5], and Giovanna Rosone[3]

[1] Department of Informatics, Systems and Communication, University of Milano-Bicocca, Milan, Italy, giulia.bernardini@unimib.it
[2] Department of Informatics, King's College London, London, UK
[huiping.chen,grigorios.loukides]@kcl.ac.uk
[3] Department of Computer Science, University of Pisa, Pisa, Italy
[conte,grossi,pisanti]@di.unipi.it, giovanna.rosone@unipi.it
[4] ERABLE Team, INRIA, Lyon, France
[5] CWI, Amsterdam, The Netherlands, solon.pissis@cwi.nl

Abstract. String data are often disseminated to support applications such as location-based service provision or DNA sequence analysis. This dissemination, however, may expose sensitive patterns that model confidential knowledge (*e.g.,* trips to mental health clinics from a string representing a user's location history). In this paper, we consider the problem of sanitizing a string by concealing the occurrences of sensitive patterns, while maintaining data utility. First, we propose a time-optimal algorithm, TFS-ALGO, to construct the shortest string preserving the order of appearance and the frequency of all non-sensitive patterns. Such a string allows accurately performing tasks based on the sequential nature and pattern frequencies of the string. Second, we propose a time-optimal algorithm, PFS-ALGO, which preserves a partial order of appearance of non-sensitive patterns but produces a much shorter string that can be analyzed more efficiently. The strings produced by either of these algorithms may reveal the location of sensitive patterns. In response, we propose a heuristic, MCSR-ALGO, which replaces letters in these strings with carefully selected letters, so that sensitive patterns are not reinstated and occurrences of spurious patterns are prevented. We implemented our sanitization approach that applies TFS-ALGO, PFS-ALGO and then MCSR-ALGO and experimentally show that it is effective and efficient.

1 Introduction

A large number of applications, in domains ranging from transportation to web analytics and bioinformatics feature data modeled as *strings*, *i.e.,* sequences of letters over some finite alphabet. For instance, a string may represent the history of visited locations of one or more individuals, with each letter corresponding to a location. Similarly, it may represent the history of search query terms of one or more web users, with letters corresponding to query terms, or a medically important part of the DNA sequence of a patient, with letters corresponding to DNA bases. Analyzing such strings is key in applications including location-based service provision, product recommendation, and DNA sequence analysis.

Therefore, such strings are often disseminated beyond the party that has collected them. For example, location-based service providers often outsource their data to data analytics companies who perform tasks such as similarity evaluation between strings [15], and retailers outsource their data to marketing agencies who perform tasks such as mining frequent patterns from the strings [16].

However, disseminating a string intact may result in the exposure of confidential knowledge, such as trips to mental health clinics in transportation data [23], query terms revealing political beliefs or sexual orientation of individuals in web data [19], or diseases associated with certain parts of DNA data [17]. Thus, it may be necessary to sanitize a string prior to its dissemination, so that confidential knowledge is not exposed. At the same time, it is important to preserve the utility of the sanitized string, so that data protection does not outweigh the benefits of disseminating the string to the party that disseminates or analyzes the string, or to the society at large. For example, a retailer should still be able to obtain actionable knowledge in the form of frequent patterns from the marketing agency who analyzed their outsourced data; and researchers should still be able to perform analyses such as identifying significant patterns in DNA sequences.

Our Model and Setting. Motivated by the discussion above, we introduce the following model which we call *Combinatorial String Dissemination* (CSD). In CSD, a party has a string W that it seeks to disseminate, while satisfying a set of *constraints* and a set of desirable *properties*. For instance, the constraints aim to capture privacy requirements and the properties aim to capture data utility considerations (*e.g.,* posed by some other party based on applications). To satisfy both, W must be transformed to a string X by applying a sequence of edit operations. The computational task is to determine this sequence of edit operations so that X satisfies the desirable properties subject to the constraints.

Under the CSD model, we consider a specific setting in which the sanitized string X must satisfy the following constraint **C1**: for an integer $k > 0$, no given length-k substring (also called pattern) modeling confidential knowledge should occur in X. We call each such length-k substring a *sensitive pattern*. We aim at finding the shortest possible string X satisfying the following desired properties: (**P1**) the order of appearance of all other length-k substrings (*non-sensitive patterns*) is the same in W and in X; and (**P2**) the frequency of these length-k substrings is the same in W and in X. The problem of constructing X in this setting is referred to as TFS (Total order, Frequency, Sanitization). Clearly, substrings of arbitrary lengths can be hidden from X by setting k equal to the length of the shortest substring we wish to hide, and then setting, for each of these substrings, any length-k substring as sensitive.

Our setting is motivated by real-world applications involving string dissemination. In these applications, a *data custodian* disseminates the sanitized version X of a string W to a *data recipient*, for the purpose of analysis (*e.g.,* mining). W contains confidential information that the data custodian needs to hide, so that it does not occur in X. Such information is specified by the data custodian based on domain expertise, as in [1,4,12,16]. At the same time, the data recipient specifies **P1** and **P2** that X must satisfy in order to be useful. These properties map directly to common data utility considerations in string analysis. By satisfying **P1**, X allows tasks based on the sequential nature of the string, such as blockwise q-gram distance computation [13], to be performed accurately. By satisfying **P2**, X allows computing the frequency of length-k substrings [21] and hence mining frequent length-k substrings with no utility loss. We require that X has minimal length so that it does not contain redundant information. For instance, the string which is constructed by concatenating all non-sensitive length-k substrings in W and separating them with a special letter that does not occur in W, satisfies **P1** and **P2** but is not the shortest possible. Such a string X will have a negative impact on the efficiency of any subsequent analysis tasks to be performed on it.

Note, existing works for sequential data sanitization (*e.g.,* [4,12,14,16,25]) or anonymization (*e.g.,* [3,5,7]) cannot be applied to our setting (see Section 7).

Our Contributions. We define the TFS problem for string sanitization and a variant of it, referred to as PFS (Partial order, Frequency, Sanitization), which aims at producing an even shorter string Y by relaxing **P1** of TFS. Our algorithms for TFS and PFS construct strings X and Y using a separator letter #, which is not contained in the alphabet of W. This prevents occurrences of sensitive patterns in X or Y. The algorithms repeat proper substrings of sensitive patterns so that the frequency of non-sensitive patterns overlapping with sensitive ones does not change. For X, we give a deterministic construction which may be easily reversible (*i.e.,* it may enable a data recipient to construct W from X), because the occurrences of # reveal the exact location of sensitive patterns. For Y, we give a construction which breaks several ties arbitrarily, thus being less easily reversible. We further address the reversibility issue by defining the MCSR (Minimum-Cost Separators Replacement) problem and designing an algorithm for dealing with it. In MCSR, we seek to replace all separators, so that the location of sensitive patterns is not revealed, while preserving data utility. We make the following specific contributions:

1. We design an algorithm for solving the TFS problem in $\mathcal{O}(kn)$ time, where n is the length of W. In fact we prove that $\mathcal{O}(kn)$ time is worst-case optimal by showing that the length of X is in $\Theta(kn)$ in the worst case. The output of the algorithm is a string X consisting of a sequence of substrings over the alphabet of W separated by # (see Example 1 below). An important feature of our algorithm, which is useful in the efficient construction of Y discussed next, is that it can be implemented to produce an $\mathcal{O}(n)$-sized representation of X with respect to W in $\mathcal{O}(n)$ time. See Section 3.

Example 1. Let $W = \texttt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\{\texttt{aaaa}, \texttt{baaa}, \texttt{bbaa}\}$. The string $X = \texttt{aabaa\#aaababbba\#baab}$ consists of three substrings over the alphabet $\{\texttt{a}, \texttt{b}\}$ separated by #. Note that no sensitive pattern occurs in X, while all non-sensitive substrings of length 4 have the same frequency in W and in X (*e.g.,* aaba appears twice) and appear in the same order in W and in X (*e.g.,* babb precedes abbb). Also, note that any shorter string than X would either create sensitive patterns or change the frequencies (*e.g.,* removing the last letter of X creates a string in which baab no longer appears). □

2. We define the PFS problem relaxing **P1** of TFS to produce shorter strings that are more efficient to analyze. Instead of a *total order* (**P1**), we require a *partial order* ($\Pi 1$) that preserves the order of appearance only for sequences of successive non-sensitive length-k substrings that overlap by $k-1$ letters. This makes sense because the order of two successive non-sensitive length-k substrings with no length-$(k-1)$ overlap has anyway been "interrupted" (by a sensitive pattern). We exploit this observation to shorten the string further. Specifically, we design an algorithm that solves PFS in the optimal $\mathcal{O}(n+|Y|)$ time, where $|Y|$ is the length of Y, using the $\mathcal{O}(n)$-sized representation of X. See Section 4.

Example 2. (Cont'd from Example 1) Recall that $W =$ `aabaaaababbbaab`. A string Y is `aaababbba#aabaab`. The order of `babb` and `abbb` is preserved in Y since they are successive, non-sensitive, and with an overlap of $k-1=3$ letters. The order of `abaa` and `aaab`, which are successive and non-sensitive, is not preserved since they do not have an overlap of $k-1=3$ letters. □

3. We define the MCSR problem, which seeks to produce a string Z, by deleting or replacing all separators in Y with letters from the alphabet of W so that: no sensitive patterns are reinstated in Z; occurrences of spurious patterns that may not be mined from W but can be mined from Z, for a given support threshold, are prevented; the distortion incurred by the replacements in Z is bounded. The first requirement is to preserve privacy and the next two to preserve data utility. We show that MCSR is NP-hard and propose a heuristic to attack it. See Section 5.

4. We implemented our combinatorial approach for sanitizing a string W (*i.e.,* all aforementioned algorithms implementing the pipeline $W \to X \to Y \to Z$) and show its effectiveness and efficiency on real and synthetic data. See Section 6.

2 Preliminaries, Problem Statements, and Main Results

Preliminaries. Let $T = T[0]T[1]\dots T[n-1]$ be a *string* of length $|T| = n$ over a finite ordered alphabet Σ of size $|\Sigma| = \sigma$. By Σ^* we denote the set of all strings over Σ. By Σ^k we denote the set of all length-k strings over Σ. For two positions i and j on T, we denote by $T[i\mathinner{.\,.}j] = T[i]\dots T[j]$ the *substring* of T that starts at position i and ends at position j of T. By ε we denote the *empty string* of length 0. A *prefix* of T is a substring of the form $T[0\mathinner{.\,.}j]$, and a suffix of T is a substring of the form $T[i\mathinner{.\,.}n-1]$. A *proper* prefix (suffix) of a string is not equal to the string itself. By $\mathrm{Freq}_V(U)$ we denote the number of occurrences of string U in string V. Given two strings U and V we say that U has a *suffix-prefix overlap* of length $\ell > 0$ with V if and only if the length-ℓ suffix of U is equal to the length-ℓ prefix of V, *i.e.,* $U[|U|-\ell\mathinner{.\,.}|U|-1] = V[0\mathinner{.\,.}\ell-1]$.

We fix a string W of length n over an alphabet $\Sigma = \{1,\dots,n^{\mathcal{O}(1)}\}$ and an integer $0 < k < n$. We refer to a length-k string or a *pattern* interchangeably. An occurrence of a pattern is uniquely represented by its starting position. Let $\mathcal{S}$ be a set of positions over $\{0,\dots,n-k\}$ with the following closure property: for every $i \in \mathcal{S}$, if there exists j such that $W[j\mathinner{.\,.}j+k-1] = W[i\mathinner{.\,.}i+k-1]$, then $j \in \mathcal{S}$. That is, if an occurrence of a pattern is in $\mathcal{S}$ all its occurrences are in $\mathcal{S}$. A substring $W[i\mathinner{.\,.}i+k-1]$ of W is called *sensitive* if and only if $i \in \mathcal{S}$. $\mathcal{S}$ is thus the set of occurrences of sensitive patterns. The difference set $\mathcal{I} = \{0, \ldots, n-k\} \setminus \mathcal{S}$ is the set of occurrences of *non-sensitive* patterns.

For any string U, we denote by $\mathcal{I}_U$ the set of occurrences of non-sensitive length-k strings over Σ. (We have that $\mathcal{I}_W = \mathcal{I}$.) We call an occurrence i the *t-predecessor* of another occurrence j in $\mathcal{I}_U$ if and only if i is the largest element in $\mathcal{I}_U$ that is less than j. This relation induces a *strict total order* on the occurrences in $\mathcal{I}_U$. We call i the *p-predecessor* of j in $\mathcal{I}_U$ if and only if i is the t-predecessor of j in $\mathcal{I}_U$ *and* $U[i \mathinner{.\,.} i+k-1]$ has a suffix-prefix overlap of length $k-1$ with $U[j \mathinner{.\,.} j+k-1]$. This relation induces a *strict partial order* on the occurrences in $\mathcal{I}_U$. We call a subset $\mathcal{J}$ of $\mathcal{I}_U$ a *t-chain* (resp., *p-chain*) if for all elements in $\mathcal{J}$ except the minimum one, their t-predecessor (resp., p-predecessor) is also in $\mathcal{J}$. For two strings U and V, chains $\mathcal{J}_U$ and $\mathcal{J}_V$ are *equivalent*, denoted by $\mathcal{J}_U \equiv \mathcal{J}_V$, if and only if $|\mathcal{J}_U| = |\mathcal{J}_V|$ and $U[u \mathinner{.\,.} u+k-1] = V[v \mathinner{.\,.} v+k-1]$, where u is the jth smallest element of $\mathcal{J}_U$ and v is the jth smallest of $\mathcal{J}_V$, for all $j \leq |\mathcal{J}_U|$.

Problem Statements and Main Results.

Problem 1 (TFS). *Given W, k, $\mathcal{S}$, and $\mathcal{I}$ construct the* shortest *string X:*
C1 *X does not contain any sensitive pattern.*
P1 *$\mathcal{I}_W \equiv \mathcal{I}_X$, i.e., the t-chains $\mathcal{I}_W$ and $\mathcal{I}_X$ are equivalent.*
P2 *$\mathit{Freq}_X(U) = \mathit{Freq}_W(U)$, for all $U \in \Sigma^k \setminus \{W[i \mathinner{.\,.} i+k-1] : i \in \mathcal{S}\}$.*

TFS requires constructing the shortest string X in which all sensitive patterns from W are concealed (**C1**), while preserving the order (**P1**) and the frequency (**P2**) of all non-sensitive patterns. Our first result is the following.

Theorem 1. *Let W be a string of length n over $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. Given $k < n$ and $\mathcal{S}$,* TFS-ALGO *solves Problem 1 in $\mathcal{O}(kn)$ time, which is worst-case optimal. An $\mathcal{O}(n)$-sized representation of X can be built in $\mathcal{O}(n)$ time.*

P1 implies **P2**, but **P1** is a strong assumption that may result in long output strings that are inefficient to analyze. We thus relax **P1** to require that the order of appearance remains the same only for sequences of successive non-sensitive length-k substrings that also overlap by $k-1$ letters (p-chains).

Problem 2 (PFS). *Given W, k, $\mathcal{S}$, and $\mathcal{I}$ construct a* shortest *string Y:*
C1 *Y does not contain any sensitive pattern.*
*Π***1** *For any p-chain $\mathcal{J}_W$ of $\mathcal{I}_W$, there is a p-chain $\mathcal{J}_Y$ of $\mathcal{I}_Y$ such that $\mathcal{J}_W \equiv \mathcal{J}_Y$.*
P2 *$\mathit{Freq}_Y(U) = \mathit{Freq}_W(U)$, for all $U \in \Sigma^k \setminus \{W[i \mathinner{.\,.} i+k-1] : i \in \mathcal{S}\}$.*

Our second result, which builds on Theorem 1, is the following.

Theorem 2. *Let W be a string of length n over $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. Given $k < n$ and $\mathcal{S}$,* PFS-ALGO *solves Problem 2 in the optimal $\mathcal{O}(n + |Y|)$ time.*

To arrive at Theorems 1 and 2, we use a special letter (separator) $\# \notin \Sigma$ when required. However, the occurrences of # may reveal the locations of sensitive patterns. We thus seek to delete or replace the occurrences of # in Y with letters from Σ. The new string Z should not reinstate any sensitive pattern. Given an integer threshold $\tau > 0$, we call pattern $U \in \Sigma^k$ a *τ-ghost* in Z if and only if $\text{Freq}_W(U) < \tau$ but $\text{Freq}_Z(U) \geq \tau$. Moreover, we seek to prevent *τ-ghost occurrences* in Z by also bounding the total *weight* of the *letter choices* we make to replace the occurrences of #. This is the MCSR problem. We show that already a restricted version of the MCSR problem, namely, the version when $k = 1$, is NP-hard via the *Multiple Choice Knapsack* (MCK) problem [20].

Theorem 3. *The* MCSR *problem is NP-hard.*

Based on this connection, we propose a non-trivial heuristic algorithm to attack the MCSR problem for the general case of an arbitrary k.

3 TFS-ALGO

We convert string W into a string X over alphabet $\Sigma \cup \{\#\}$, $\# \notin \Sigma$, by reading the letters of W, from left to right, and appending them to X while enforcing the following two rules:

R1: When the last letter of a sensitive substring U is read from W, we append # to X (essentially replacing this last letter of U with #). Then, if V is the longest proper prefix of the succeeding non-sensitive substring (in the t-predecessor order), we append the longest proper suffix V of U right after #.
R2: When the $k - 1$ letters before # are the same as the $k - 1$ letters after #, we remove # and the $k - 1$ succeeding letters (inspect Fig. 1).

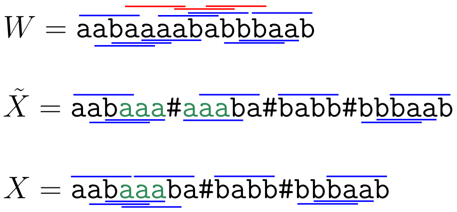

W = aabaaaababbbbaab
$\tilde{X}$ = aabaaa#aaaba#babb#bbbaab
X = aabaaaba#babb#bbbaab

Fig. 1: Sensitive patterns are overlined in red; non-sensitive are under- or over-lined in blue; $\tilde{X}$ is obtained by applying **R1**; and X by applying **R1** and **R2**. In green we highlight an overlap of $k - 1 = 3$ letters. Note that substring `aaaababbb`, whose length is greater than k, is also not occurring in X.

R1 prevents U from occurring in X, and **R2** reduces the length of X (*i.e.*, allows to protect sensitive patterns with fewer extra letters). Both rules leave unchanged the order and frequencies of non-sensitive patterns. It is crucial to observe that applying the idea behind **R2** on more than $k - 1$ letters would decrease the frequency of some pattern, while applying it on fewer than $k - 1$ letters would create new patterns. Thus, we need to consider just **R2** *as-is*.

Let C be an array of size n that stores the occurrences of sensitive and non-sensitive patterns: $C[i] = 1$ if $i \in \mathcal{S}$ and $C[i] = 0$ if $i \in \mathcal{I}$. For technical reasons we set the last $k - 1$ values in C equal to $C[n - k]$; *i.e.*, $C[n - k + 1] := \ldots := C[n - 1] := C[n - k]$. Note that C is constructible from $\mathcal{S}$ in $\mathcal{O}(n)$ time. Given C and $k < n$, TFS-ALGO efficiently constructs X by implementing **R1** and **R2** concurrently as opposed to implementing **R1** and then **R2** (see the proof of Lemma 1 for details of the workings of TFS-ALGO and Fig. 1 for an example). We next show that string X enjoys several properties.

Lemma 1. *Let W be a string of length n over Σ. Given $k < n$ and array C,* TFS-ALGO *constructs the shortest string X such that the following hold:*

1. *There exists no $W[i \mathinner{.\,.} i + k - 1]$ with $C[i] = 1$ occurring in X (**C1**).*

2. *$\mathcal{I}_W \equiv \mathcal{I}_X$, i.e., the order of substrings $W[i\mathinner{.\,.}i+k-1]$, for all i such that $C[i]=0$, is the same in W and in X; conversely, the order of all substrings $U \in \Sigma^k$ of X is the same in X and in W (**P1**).*
3. *$Freq_X(U) = Freq_W(U)$, for all $U \in \Sigma^k \setminus \{W[i\mathinner{.\,.}i+k-1] : C[i]=1\}$ (**P2**).*
4. *The occurrences of letter # in X are at most $\lfloor \frac{n-k+1}{2} \rfloor$ and they are at least k positions apart (**P3**).*
5. *$0 \le |X| \le \lceil \frac{n-k+1}{2} \rceil \cdot k + \lfloor \frac{n-k+1}{2} \rfloor$ and these bounds are tight (**P4**).*

```
TFS-ALGO($W \in \Sigma^n, C, k, \# \notin \Sigma$)
1  $X \leftarrow \varepsilon$; $j \leftarrow |W|$; $\ell \leftarrow 0$;
2  $j \leftarrow \min\{i|C[i]=0\}$; /* $j$ is the leftmost pos of a non-sens. pattern */
3  if $j+k-1 < |W|$ then /* Append the first non-sens. pattern to $X$ */
4      $X[0\mathinner{.\,.}k-1] \leftarrow W[j\mathinner{.\,.}j+k-1]$; $j \leftarrow j+k$; $\ell \leftarrow \ell+k$;
5  while $j < |W|$ do /* Examine two consecutive patterns */
6      $p \leftarrow j-k$; $c \leftarrow p+1$;
7      if $C[p]=C[c]=0$ then /* If both are non-sens., append the last letter of the leftmost one to $X$ */
8          $X[\ell] \leftarrow W[j]$; $\ell \leftarrow \ell+1$; $j \leftarrow j+1$;
9      if $C[p]=0 \wedge C[c]=1$ then /* If the rightmost is sens., mark it and advance $j$ */
10         $f \leftarrow c$; $j \leftarrow j+1$;
11     if $C[p]=C[c]=1$ then $j \leftarrow j+1$; /* If both are sens., advance $j$ */
12     if $C[p]=1 \wedge C[c]=0$ then /* If the leftmost is sens. and the rightmost is not */
13         if $W[c\mathinner{.\,.}c+k-2] = W[f\mathinner{.\,.}f+k-2]$ then /* If the last marked sens. pattern and the current non-sens. overlap by $k-1$, append the last letter of the latter to $X$ */
14             $X[\ell] \leftarrow W[j]$; $\ell \leftarrow \ell+1$; $j \leftarrow j+1$;
15         else /* Else append $\#$ and the $(k-1)$-length suffix of the current non-sens. pattern to $X$ */
16             $X[\ell] \leftarrow \#$; $\ell \leftarrow \ell+1$;
17             $X[\ell\mathinner{.\,.}\ell+k-1] \leftarrow W[j-k+1\mathinner{.\,.}j]$; $\ell \leftarrow \ell+k$; $j \leftarrow j+1$;
18 report $X$
```

Proof. Proofs of **C1** and **P1-P4** can be found in the appendix. We prove here that X has minimal length. Let X_j be the prefix of string X obtained by processing the first j letters of string W. Let $j_{\min} = \min\{i|C[i]=0\}+k$. We will proceed by induction on j, claiming that X_j is the shortest string such that **C1** and **P1-P4** hold for $W[0\mathinner{.\,.}j]$, $\forall\, j_{\min} \le j \le |W|-1$. We call such a string *optimal*.

Base case: $j = j_{\min}$. By Lines 3-4 of TFS-ALGO, X_j is equal to the first non-sensitive length-k substring of W, and it is clearly the shortest string such that **C1** and **P1-P4** hold for $W[0\mathinner{.\,.}j]$.

Inductive hypothesis and step: X_{j-1} is optimal for $j > j_{\min}$. If $C[j-k] = C[j-k+1] = 0$, $X_j = X_{j-1}W[j]$ and this is clearly optimal. If $C[j-k+1]=1$, $X_j = X_{j-1}$ thus still optimal. Finally, if $C[j-k]=1$ and $C[j-k+1]=0$ we have two subcases: if $W[f\mathinner{.\,.}f+k-2] = W[j-k+1\mathinner{.\,.}j-1]$ then $X_j = X_{j-1}W[j]$, and once again X_j is evidently optimal. Otherwise, $X_j = X_{j-1}\#W[j-k+1\mathinner{.\,.}j]$.

Suppose by contradiction that there exists a shorter X'_j such that **C1** and **P1**-**P4** still hold: either drop # or append less than k letters after #. If we appended less than k letters after #, since TFS-ALGO will not read $W[j]$ ever again, **P2**-**P3** would be violated, as an occurrence of $W[j-k+1 \mathinner{.\,.} j]$ would be missed. Without #, the last k letters of $X_{j-1}W[j-k+1]$ would violate either **C1** or **P1** and **P2** (since we suppose $W[f \mathinner{.\,.} f+k-2] \neq W[j-k+1 \mathinner{.\,.} j-1]$). Then X_j is optimal. □

Theorem 1. *Let W be a string of length n over $\Sigma = \{1,\ldots,n^{\mathcal{O}(1)}\}$. Given $k<n$ and $\mathcal{S}$,* TFS-ALGO *solves Problem 1 in $\mathcal{O}(kn)$ time, which is worst-case optimal. An $\mathcal{O}(n)$-sized representation of X can be built in $\mathcal{O}(n)$ time.*

Proof. For the first part inspect TFS-ALGO. Lines 2-4 can be realized in $\mathcal{O}(n)$ time. The *while* loop in Line 5 is executed no more than n times, and every operation inside the loop takes $\mathcal{O}(1)$ time except for Line 13 and Line 17 which take $\mathcal{O}(k)$ time. Correctness and optimality follow directly from Lemma 1 (**P4**).

For the second part, we assume that X is represented by W and a sequence of pointers $[i,j]$ to W interleaved (if necessary) by occurrences of #. In Line 17, we can use an interval $[i,j]$ to represent the length-k substring of W added to X. In all other lines (Lines 4, 8 and 14) we can use $[i,i]$ as one letter is added to X per one letter of W. By Lemma 1 we can have at most $\lfloor \frac{n-k+1}{2} \rfloor$ occurrences of letter #. The check at Line 13 can be implemented in constant time after linear-time pre-processing of W for longest common extension queries [9]. All other operations take in total linear time in n. Thus there exists an $\mathcal{O}(n)$-sized representation of X and it is constructible in $\mathcal{O}(n)$ time. □

4 PFS-ALGO

Lemma 1 tells us that X is the shortest string satisfying constraint **C1** and properties **P1**-**P4**. If we were to drop **P1** and employ the partial order $\Pi 1$ (see Problem 2), the length of $X = X_1\#\ldots\#X_N$ would not always be minimal: if a *permutation* of the strings $X_1,\ldots,X_N$ contains pairs X_i, X_j with a suffix-prefix overlap of length $\ell = k-1$, we may further apply **R2**, obtaining a shorter string while still satisfying $\Pi 1$.

To find such a permutation efficiently and construct a shorter string Y from W, we propose PFS-ALGO. The crux of our algorithm is an efficient method to solve a variant of the classic NP-complete *Shortest Common Superstring* (SCS) problem [11]. Specifically our algorithm: (I) Computes the string X using Theorem 1. (II) Constructs a collection $\mathcal{B}'$ of strings, each of two symbols (two identifiers): the first (resp., second) symbol of the ith element of $\mathcal{B}'$ is a unique identifier of the string corresponding to the ℓ-length prefix (resp., suffix) of the ith element of $\mathcal{B} = \{X_1,\ldots,X_N\}$. (III) Computes a shortest string containing every element in $\mathcal{B}'$ as a distinct substring. (IV) Constructs Y by mapping back each element to its distinct substring in $\mathcal{B}$. If there are multiple possible shortest strings, one is selected arbitrarily.

Example 3 (Illustration of the workings of PFS-ALGO*).* Let $\ell = k-1 = 3$ and

$$X = \texttt{aabbc\#bccaab\#bbca\#aaabac\#aabcbbc}.$$

The collection $\mathcal{B}$ is $\texttt{aabbc}, \texttt{bccaab}, \texttt{bbca}, \texttt{aaabac}, \texttt{aabcbbc}$, and the collection $\mathcal{B}'$ is $\texttt{24}, \texttt{62}, \texttt{45}, \texttt{13}, \texttt{24}$ (id of prefix · id of suffix). A shortest string containing all elements of $\mathcal{B}'$ as distinct substrings is: $\texttt{13} \cdot \texttt{24} \cdot \texttt{6245}$ (obtained by permuting the original string as $\texttt{13}, \texttt{24}, \texttt{62}, \texttt{24}, \texttt{45}$ then applying **R2** twice). This shortest string is mapped back to the solution $Y = \texttt{aaabac\#aabbc\#bccaabcbbca}$. For example, $\texttt{13}$ is mapped back to $\texttt{aaabac}$. Note, Y contains two occurrences of # and has length 24, while X contains 4 occurrences of # and has length 32. □

We now present the details of PFS-ALGO. We first introduce the *Fixed-Overlap Shortest String with Multiplicities* (FO-SSM) problem: Given a *collection* $\mathcal{B}$ of strings $B_1, \ldots, B_{|\mathcal{B}|}$ and an integer ℓ, with $|B_i| > \ell$, for all $1 \leq i \leq |\mathcal{B}|$, FO-SSM seeks to find a shortest string containing each element of $\mathcal{B}$ as a distinct substring using the following operations on any pair of strings B_i, B_j:

1. $\texttt{concat}(B_i, B_j) = B_i \cdot B_j$;
2. $\ell\text{-}\texttt{merge}(B_i, B_j) = B_i[0 \mathinner{..} |B_i| - \ell]B_j[0 \mathinner{..} |B_j| - 1] = B_i[0 \mathinner{..} |B_i| - \ell] \cdot B_j$.

Any solution to FO-SSM with $\ell := k - 1$ and $\mathcal{B} := X_1, \ldots, X_N$ implies a solution to the PFS problem, because $|X_i| > k - 1$ for all i's (see Lemma 1, **P3**)

The FO-SSM problem is a variant of the SCS problem. In the SCS problem, we are given a *set* of strings and we are asked to compute the shortest common superstring of the elements of this set. The SCS problem is known to be NP-Complete, even for binary strings [11]. However, if all strings are of length two, the SCS problem admits a linear-time solution [11]. We exploit this crucial detail positively to show a linear-time solution to the FO-SSM problem in Lemma 3. In order to arrive to this result, we first adapt the SCS linear-time solution of [11] to our needs (see Lemma 2) and plug this solution to Lemma 3.

Lemma 2. *Let $\mathcal{Q}$ be a collection of q strings, each of length two, over an alphabet $\Sigma = \{1, \ldots, (2q)^{\mathcal{O}(1)}\}$. We can compute a shortest string containing every element of $\mathcal{Q}$ as a distinct substring in $\mathcal{O}(q)$ time.*

Proof. We sort the elements of $\mathcal{Q}$ lexicographically in $\mathcal{O}(q)$ time using radixsort. We also replace every letter in these strings with their *lexicographic rank* from $\{1, \ldots, 2q\}$ in $\mathcal{O}(q)$ time using radixsort. In $\mathcal{O}(q)$ time we construct the de Bruijn multigraph G of these strings [6]. Within the same time complexity, we find all nodes v in G with in-degree, denoted by $\text{IN}(v)$, smaller than out-degree, denoted by $\text{OUT}(v)$. We perform the following two steps:

Step 1: While there exists a node v in G with $\text{IN}(v) < \text{OUT}(v)$, we start an arbitrary path (with possibly repeated nodes) from v, traverse consecutive edges and delete them. Each time we delete an edge, we update the in- and out-degree of the affected nodes. We stop traversing edges when a node v' with $\text{OUT}(v') = 0$ is reached: whenever $\text{IN}(v') = \text{OUT}(v') = 0$, we also delete v' from G. Then, we add the traversed path $p = v \ldots v'$ to a set $\mathcal{P}$ of paths. The path can contain the same node v more than once. If G is empty we halt. Proceeding this way, there are no two elements p_1 and p_2 in $\mathcal{P}$ such that p_1 starts with v and p_2 ends with v; thus this path decomposition is minimal. If G is not empty at the end, by construction, it consists of only cycles.

Step 2: While G is not empty, we perform the following. If there exists a cycle c that *intersects* with any path p in $\mathcal{P}$ we splice c with p, update p with the result of splicing, and delete c from G. This operation can be efficiently implemented by maintaining an array A of size $2q$ of linked lists over the paths in $\mathcal{P}$: $A[\alpha]$ stores a list of pointers to all occurrences of letter α in the elements of $\mathcal{P}$. Thus in constant time per node of c we check if any such path p exists in $\mathcal{P}$ and splice the two in this case. If no such path exists in $\mathcal{P}$, we add to $\mathcal{P}$ any of the path-linearizations of the cycle, and delete the cycle from G. After each change to $\mathcal{P}$, we update A and delete every node u with $\mathrm{IN}(u) = \mathrm{OUT}(u) = 0$ from G.

The correctness of this algorithm follows from the fact that $\mathcal{P}$ is a minimal path decomposition of G. Thus any concatenation of paths in $\mathcal{P}$ represents a shortest string containing all elements in $\mathcal{Q}$ as distinct substrings. □

Omitted proofs of Lemmas 3 and 4 can be found in the appendix.

Lemma 3. *Let $\mathcal{B}$ be a collection of strings over an alphabet $\Sigma = \{1, \ldots, ||\mathcal{B}||^{\mathcal{O}(1)}\}$. Given an integer ℓ, the FO-SSM problem for $\mathcal{B}$ can be solved in $\mathcal{O}(||\mathcal{B}||)$ time.*

Thus, PFS-ALGO applies Lemma 3 on $\mathcal{B} := X_1, \ldots, X_N$ with $\ell := k - 1$ (recall that $X_1\# \ldots \#X_N = X$). Note that each time the `concat` operation is performed, it also places the letter # in between the two strings.

Lemma 4. *Let W be a string of length n over an alphabet Σ. Given $k < n$ and array C,* PFS-ALGO *constructs a shortest string Y with **C1**, $\Pi 1$, and **P2-P4**.*

Theorem 2. *Let W be a string of length n over $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$. Given $k < n$ and $\mathcal{S}$,* PFS-ALGO *solves Problem 2 in the optimal $\mathcal{O}(n + |Y|)$ time.*

Proof. We compute the $\mathcal{O}(n)$-sized representation of string X with respect to W described in the proof of Theorem 1. This can be done in $\mathcal{O}(n)$ time. If $X \in \Sigma^*$, then we construct and return $Y := X$ in time $\mathcal{O}(|Y|)$ from the representation. If $X \in (\Sigma \cup \{\#\})^*$, implying $|Y| \leq |X|$, we compute the LCP data structure of string W in $\mathcal{O}(n)$ time [9]; and implement Lemma 3 in $\mathcal{O}(n)$ time by avoiding to read string X explicitly: we rather rename $X_1, \ldots, X_N$ to a collection of two-letter strings by employing the LCP information of W directly. We then construct and report Y in time $\mathcal{O}(|Y|)$. Correctness follows directly from Lemma 4. □

5 The MCSR Problem and MCSR-ALGO

The strings X and Y, constructed by TFS-ALGO and PFS-ALGO, respectively, may contain the separator #, which reveals information about the location of the sensitive patterns in W. Specifically, a malicious data recipient can go to the position of a # in X and “undo” Rule **R1** that has been applied by TFS-ALGO, removing # and the $k-1$ letters after # from X. The result will be an occurrence of the sensitive pattern. For example, applying this process to the first # in X shown in Fig. 1, results in recovering the sensitive pattern `abab`. A similar attack is possible on the string Y produced by PFS-ALGO, although it is hampered by the fact that substrings within two consecutive #s in X often swap places in Y.

To address this issue, we seek to construct a new string Z, in which #s are either deleted or replaced by letters from Σ. To preserve privacy, we require separator replacements not to reinstate sensitive patterns. To preserve data utility, we favor separator replacements that have a small cost in terms of occurrences of τ-ghosts (patterns with frequency less than τ in W and at least τ in Z) and incur a bounded level of distortion in Z, as defined below. This is the MCSR problem, a restricted version of which is presented in Problem 3. The restricted version is referred to as MCSR$_{k=1}$ and differs from MCSR in that it uses $k = 1$ for the pattern length instead of an arbitrary value $k > 0$. MCSR$_{k=1}$ is presented next for simplicity and because it is used in the proof of Lemma 5 (see the appendix for the proof). Lemma 5 implies Theorem 3.

Problem 3 (MCSR$_{k=1}$). *Given a string Y over an alphabet $\Sigma \cup \{\#\}$ with $\delta > 0$ occurrences of letter $\#$, and parameters τ and θ, construct a new string Z by substituting the δ occurrences of $\#$ in Y with letters from Σ, such that:*

$$\text{(I)} \sum_{\substack{i:Y[i]=\#,\ \mathit{Freq}_Y(Z[i])<\tau \\ \mathit{Freq}_Z(Z[i])\geq\tau}} \mathit{Ghost}(i, Z[i]) \textit{ is minimum, and } \text{(II)} \sum_{i:Y[i]=\#} \mathit{Sub}(i, Z[i]) \leq \theta.$$

The cost of τ-ghosts is captured by a function *Ghost*. This function assigns a cost to an occurrence of a τ-ghost, which is caused by a separator replacement at position i, and is specified based on domain knowledge. For example, with a cost equal to 1 for each gained occurrence of each τ-ghost, we penalize more heavily a τ-ghost with frequency much below τ in Y and the penalty increases with the number of gained occurrences. Moreover, we may want to penalize positions towards the end of a temporally ordered string, to avoid spurious patterns that would be deemed important in applications based on time-decaying models [8].

The replacement distortion is captured by a function *Sub* which assigns a weight to a letter that could replace a # and is specified based on domain knowledge. The maximum allowable replacement distortion is θ. Small weights favor the replacement of separators with desirable letters (*e.g.,* letters that reinstate non-sensitive frequent patterns) and letters that reinstate sensitive patterns are assigned a weight larger than θ that prohibits them from replacing a #. Similarly, weights larger than θ are assigned to letters which would lead to implausible patterns [14] if they replaced #s.

Lemma 5. *The MCSR$_{k=1}$ problem is NP-hard.*

Theorem 3. *The* MCSR *problem is NP-hard.*

MCSR-ALGO. Our MCSR-ALGO is a non-trivial heuristic that exploits the connection of the MCSR and MCK [20] problems and works by:
(I) Constructing the set of all candidate τ-ghost patterns (*i.e.,* length-k strings over Σ with frequency below τ in Y that can have frequency at least τ in Z).
(II) Creating an instance of MCK from an instance of MCSR. For this, we map the ith occurrence of # to a class C_i in MCK and each possible replacement of the occurrence with a letter j to a different item in C_i. Specifically, we consider all possible replacements with letters in Σ and also a replacement with the empty string, which models deleting (instead of replacing) the ith occurrence of #. In addition, we set the costs and weights that are input to MCK as follows. The cost

for replacing the ith occurrence of # with the letter j is set to the sum of the Ghost function for all candidate τ-ghost patterns when the ith occurrence of # is replaced by j. That is, we make the worst-case assumption that the replacement forces all candidate τ-ghosts to become τ-ghosts in Z. The weight for replacing the ith occurrence of # with letter j is set to $\text{Sub}(i, j)$.
(III) Solving the instance of MCK and translating the solution back to a (possibly suboptimal) solution of the MCSR problem. For this, we replace the ith occurrence of # with the letter corresponding to the element chosen by the MCK algorithm from class C_i, and similarly for each other occurrence of #. If the instance has no solution (*i.e.,* no possible replacement can hide the sensitive patterns), MCSR-ALGO reports that Z cannot be constructed and terminates.

Lemma 6 below states the running time of MCSR-ALGO (see the appendix for the proof on an efficient implementation of this algorithm).

Lemma 6. MCSR-ALGO *runs in* $\mathcal{O}(|Y| + k\delta\sigma + \mathcal{T}(\delta, \sigma))$ *time, where* $\mathcal{T}(\delta, \sigma)$ *is the running time of the MCK algorithm for* δ *classes with* $\sigma + 1$ *elements each.*

6 Experimental Evaluation

We evaluate our approach, referred to as TPM, in terms of *data utility* and *efficiency*. Given a string W over Σ, TPM sanitizes W by applying TFS-ALGO, PFS-ALGO, and then MCSR-ALGO, which uses the $\mathcal{O}(\delta\sigma\theta)$-time algorithm of [20] for solving the MCK instances. The final output is a string Z over Σ.
Experimental Setup and Data. We do not compare TPM against existing methods, because they are not alternatives to our approach (see Section 7). Instead, we compared against a greedy baseline referred to as BA.

BA initializes its output string Z_{BA} to W and then considers each sensitive pattern R in Z_{BA}, from left to right. For each R, it replaces the letter r of R that has the largest frequency in Z_{BA} with another letter r' that is not contained in R and has the smallest frequency in Z_{BA}, breaking all ties arbitrarily. If no such r' exists, r is replaced by # to ensure that a solution is produced (even if it may reveal the location of a sensitive pattern). Each replacement removes the occurrence of R and aims to prevent τ-ghost occurrences by selecting an r' that will not substantially increase the frequency of patterns overlapping with R.

We considered the following publicly available datasets used in [1,12,14,16]: Oldenburg (OLD), Trucks (TRU), MSNBC (MSN), the complete genome of *Escherichia coli* (DNA), and synthetic data (uniformly random strings, the largest of which is referred to as SYN). See Table 1 for the characteristics of these datasets and the parameter values used in experiments, unless stated otherwise.

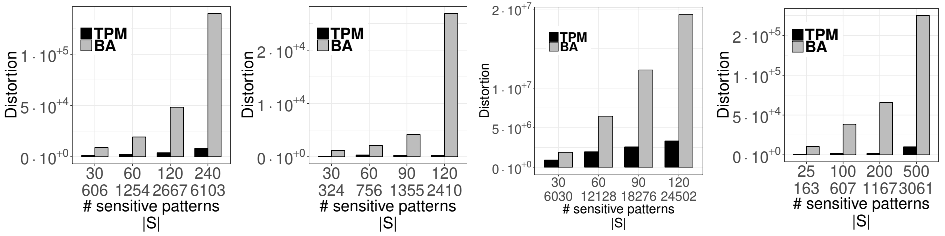
Dataset	**Data domain**	**Length** n	**Alphabet size** $\lvert\Sigma\rvert$	**# sensitive patterns**	**# sensitive positions** $\lvert\mathcal{S}\rvert$	**Pattern length** k
OLD	Movement	85,563	100	$[30, 240]$ (**60**)	$[600, 6103]$	$[3, 7]$ (**4**)
TRU	Transportation	5,763	100	$[30, 120]$ (**10**)	$[324, 2410]$	$[2, 5]$ (**4**)
MSN	Web	4,698,764	17	$[30, 120]$ (**60**)	$[6030, 320480]$	$[3, 8]$ (**4**)
DNA	Genomic	4,641,652	4	$[25, 500]$ (**100**)	$[163, 3488]$	$[5, 15]$ (**13**)
SYN	Synthetic	20,000,000	10	$[10, 1000]$ (**1000**)	$[10724, 20171]$	$[3, 6]$ (**6**)

Table 1: Characteristics of datasets and values used (default values are in bold).

The sensitive patterns were selected randomly among the frequent length-k substrings at minimum support τ following [12,14,16]. We used the fairly low values $\tau = 10$, $\tau = 20$, $\tau = 200$, and $\tau = 20$ for TRU, OLD, MSN, and DNA, respectively, to have a wider selection of sensitive patterns. We also used a uniform cost of 1 for every occurrence of each τ-ghost, a weight of 1 (resp., ∞) for each letter replacement that does not (resp., does) create a sensitive pattern, and we further set $\theta = \delta$. This setup treats all candidate τ-ghost patterns and all candidate letters for replacement uniformly, to facilitate a fair comparison with BA which cannot distinguish between τ-ghost candidates or favor specific letters.

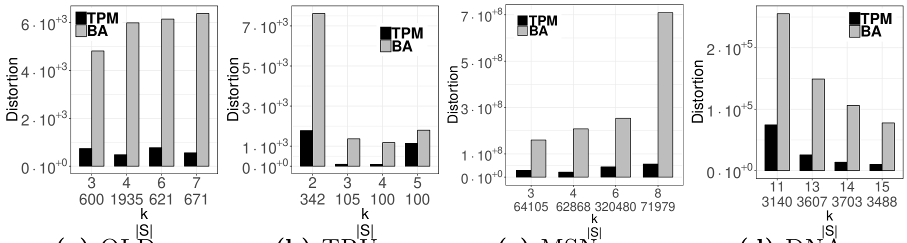
To capture the utility of sanitized data, we used the *(frequency) distortion* measure $\sum_U (\text{Freq}_W(U) - \text{Freq}_Z(U))^2$, where $U \in \Sigma^k$ is a non-sensitive pattern. The distortion measure quantifies changes in the frequency of non-sensitive patterns with low values suggesting that Z remains useful for tasks based on pattern frequency (*e.g.,* identifying motifs corresponding to functional or conserved DNA [21]). We also measured the number of τ-ghost and τ-lost patterns in Z following [12,14,16], where a pattern U is τ*-lost* in Z if and only if $\text{Freq}_W(U) \geq \tau$ but $\text{Freq}_Z(U) < \tau$. That is, τ-lost patterns model knowledge that can no longer be mined from Z but could be mined from W, whereas τ-ghost patterns model knowledge that can be mined from Z but not from W. A small number of τ-lost/ghost patterns suggests that frequent pattern mining can be accurately performed on Z [12,14,16]. Unlike BA, by design TPM *does not* incur any τ-lost pattern, as TFS-ALGO and PFS-ALGO preserve frequencies of nonsensitive patterns, and MCSR-ALGO can only increase pattern frequencies.

All experiments ran on an Intel Xeon E5-2640 at 2.66GHz with 16GB RAM. Our source code, written in `C++`, is available at `https://bitbucket.org/stringsanitization`. The results have been averaged over 10 runs.



(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

Fig. 2: Distortion vs. number of sensitive patterns and their total number $|\mathcal{S}|$ of occurrences in W(first two lines on the X axis).



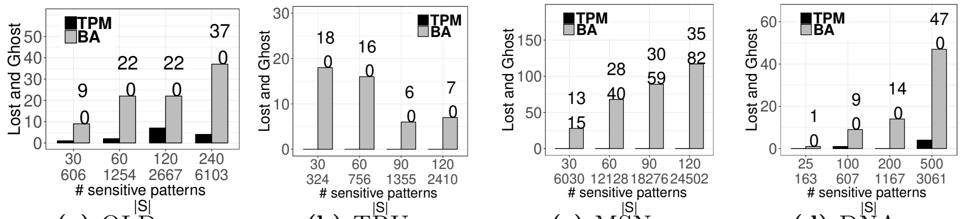
(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

Fig. 3: Distortion vs. length of sensitive patterns k (and $|\mathcal{S}|$).

Data Utility. We first demonstrate that TPM incurs *very low distortion*, which implies high utility for tasks based on the frequency of patterns (*e.g.,* [21]). Fig. 2

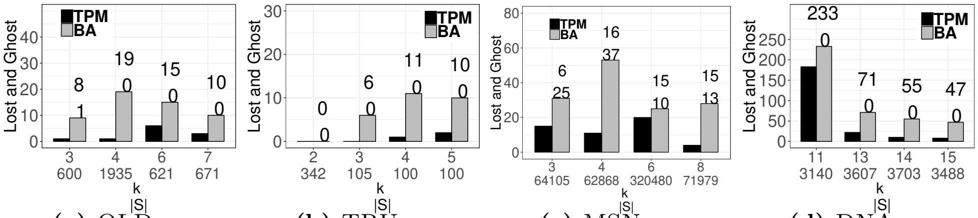
shows that, for varying number of sensitive patterns, TPM incurred on average 18.4 (and up to 95) times lower distortion than BA over all experiments. Also, Fig. 2 shows that TPM remains effective even in challenging settings, with many sensitive patterns (*e.g.,* the last point in Fig. 2b where about 42% of the positions in W are sensitive). Fig. 3 shows that, for varying k, TPM caused on average 7.6 (and up to 14) times lower distortion than BA over all experiments.

Next, we demonstrate that TPM permits *accurate frequent pattern mining*: Fig. 4 shows that TPM led to no τ-lost or τ-ghost patterns for the TRU and MSN datasets. This implies no utility loss for mining frequent length-k substrings with threshold τ. In all other cases, the number of τ-ghosts was on average 6 (and up to 12) times smaller than the total number of τ-lost and τ-ghost patterns for BA. BA performed poorly (*e.g.,* up to 44% of frequent patterns became τ-lost for TRU and 27% for DNA). Fig. 5 shows that, for varying k, TPM led to on average 5.8 (and up to 19) times fewer τ-lost/ghost patterns than BA. BA performed poorly (*e.g.,* up to 98% of frequent patterns became τ-lost for DNA).



(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

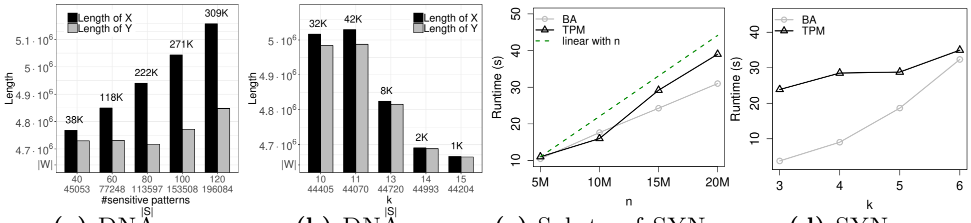
Fig. 4: Total number of τ-lost and τ-ghost patterns vs. number of sensitive patterns (and $|\mathcal{S}|$). $\frac{x}{y}$ on the top of each bar for BA denotes x τ-lost and y τ-ghost patterns.



(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

Fig. 5: Total number of τ-lost and τ-ghost patterns vs. length of sensitive patterns k (and $|\mathcal{S}|$). $\frac{x}{y}$ on the top of each bar for BA denotes x τ-lost and y τ-ghost patterns.

We also demonstrate that PFS-ALGO reduces the length of the output string X of TFS-ALGO substantially, creating a string Y that contains *less redundant information* and allows for more efficient analysis. Fig. 6a shows the length of X and of Y and their difference for $k = 5$. Y was much shorter than X and its length decreased with the number of sensitive patterns, since more substrings had a suffix-prefix overlap of length $k - 1 = 4$ and were removed (see Section 4). Interestingly, the length of Y was close to that of W (the string before sanitization). A larger k led to less substantial length reduction as shown in Fig. 6b (but still few thousand letters were removed), since it is less likely for long substrings of sensitive patterns to have an overlap and be removed.



(a) DNA **(b)** DNA **(c)** Substr. of SYN **(d)** SYN

Fig. 6: Length of X and Y (output of TFS-ALGO and PFS-ALGO, resp.) for varying: (a) number of sensitive patterns (and $|\mathcal{S}|$), (b) length of sensitive patterns k (and $|\mathcal{S}|$). On the top of each pair of bars we plot $|X| - |Y|$. Runtime on synthetic data for varying: (c) length n of string and (d) length k of sensitive patterns. Note that $|Y| = |Z|$.

Efficiency. We finally measured the runtime of TPM using prefixes of the synthetic string SYN whose length n is 20 million letters. Fig. 6c (resp., Fig. 6d) shows that TPM scaled linearly with n (resp., k), as predicted by our analysis in Section 5 (TPM takes $\mathcal{O}(n + |Y| + k\delta\sigma + \delta\sigma\theta) = \mathcal{O}(kn + k\delta\sigma + \delta\sigma\theta)$ time, since the algorithm of [20] was used for MCK instances). In addition, TPM is efficient, with a runtime similar to that of BA and less than 40 seconds for SYN.

7 Related Work

Data sanitization (*a.k.a.* knowledge hiding) aims at concealing patterns modeling confidential knowledge by limiting their frequency, so that they are not easily mined from the data. Existing methods are applied to: (I) a *collection* of set-valued data (transactions) [24] or spatiotemporal data (trajectories) [1]; (II) a *collection* of sequences [12,14]; or (III) a *single* sequence [4,16,25]. Yet, none of these methods follows our CSD setting: Methods in category I are not applicable to string data, and those in categories II and III do not have guarantees on privacy-related constraints [25] or on utility-related properties [12,14,4,16]. Specifically, unlike our approach, [25] cannot guarantee that all sensitive patterns are concealed (constraint **C1**), while [12,14,4,16] do not guarantee the satisfaction of utility properties (*e.g.,* $\Pi 1$ and **P2**).

Anonymization aims to prevent the disclosure of individuals' identity and/or information that individuals are not willing to be associated with [3,10]. Anonymization works such as [3,5,7] are thus not alternatives to our work (see the appendix).

8 Conclusion

In this paper, we introduced the Combinatorial String Dissemination model. The focus of this model is on *guaranteeing* privacy-utility trade-offs (*e.g.,* **C1** *vs.* $\Pi 1$ and **P2**). We defined a problem (TFS) which seeks to produce the shortest string that preserves the order of appearance and the frequency of all non-sensitive patterns; and a variant (PFS) that preserves a partial order and the frequency of the non-sensitive patterns but produces a shorter string. We developed two time-optimal algorithms, TFS-ALGO and PFS-ALGO, for the problem and its variant, respectively. We also developed MCSR-ALGO, a heuristic that prevents the disclosure of the location of sensitive patterns from the outputs of TFS-ALGO and PFS-ALGO. Our experiments show that sanitizing a string by TFS-ALGO, PFS-ALGO and then MCSR-ALGO is effective and efficient.

Acknowledgments. HC is supported by a CSC scholarship. GR and NP are partially supported by MIUR-SIR project CMACBioSeq grant n. RBSI146R5L. We acknowledge the use of the Rosalind HPC cluster hosted by King's College London.

References

1. Abul, O., Bonchi, F., Giannotti, F.: Hiding sequential and spatiotemporal patterns. TKDE **22**(12), 1709–1723 (2010)
2. Aggarwal, C.C., Yu, P.S.: On anonymization of string data. In: SDM. pp. 419–424 (2007)
3. Aggarwal, C.C., Yu, P.S.: A framework for condensation-based anonymization of string data. DMKD **16**(3), 251–275 (2008)
4. Bonomi, L., Fan, L., Jin, H.: An information-theoretic approach to individual sequential data sanitization. In: WSDM. pp. 337–346 (2016)
5. Bonomi, L., Xiong, L.: A two-phase algorithm for mining sequential patterns with differential privacy. In: CIKM. pp. 269–278 (2013)
6. Cazaux, B., Lecroq, T., Rivals, E.: Linking indexing data structures to de Bruijn graphs: Construction and update. J. Comput. Syst. Sci. (2016)
7. Chen, R., Acs, G., Castelluccia, C.: Differentially private sequential data publication via variable-length n-grams. In: CCS. pp. 638–649 (2012)
8. Cormode, G., Korn, F., Tirthapura, S.: Exponentially decayed aggregates on data streams. In: ICDE. pp. 1379–1381 (2008)
9. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. Cambridge University Press (2007)
10. Dwork, C., McSherry, F., Nissim, K., Smith, A.: Calibrating noise to sensitivity in private data analysis. In: TCC. pp. 265–284 (2006)
11. Gallant, J., Maier, D., Storer, J.A.: On finding minimal length superstrings. J. Comput. Syst. Sci. **20**(1), 50–58 (1980)
12. Gkoulalas-Divanis, A., Loukides, G.: Revisiting sequential pattern hiding to enhance utility. In: KDD. pp. 1316–1324 (2011)
13. Grossi, R., Iliopoulos, C.S., Mercas, R., Pisanti, N., Pissis, S.P., Retha, A., Vayani, F.: Circular sequence comparison: algorithms and applications. AMB **11**, 12 (2016)
14. Gwadera, R., Gkoulalas-Divanis, A., Loukides, G.: Permutation-based sequential pattern hiding. In: ICDM. pp. 241–250 (2013)
15. Liu, A., Zhengy, K., Liz, L., Liu, G., Zhao, L., Zhou, X.: Efficient secure similarity computation on encrypted trajectory data. In: ICDE. pp. 66–77 (2015)
16. Loukides, G., Gwadera, R.: Optimal event sequence sanitization. In: SDM. pp. 775–783 (2015)
17. Malin, B., Sweeney, L.: Determining the identifiability of DNA database entries. In: AMIA. pp. 537–541 (2000)
18. Monreale, A., Pedreschi, D., Pensa, R.G., Pinelli, F.: Anonymity preserving sequential pattern mining. Artif. Intell. Law **22**(2), 141–173 (2014)
19. Narayanan, A., Shmatikov, V.: Robust de-anonymization of large sparse datasets. In: S&P. pp. 111–125 (2008)
20. Pissinger, D.: A minimal algorithm for the multiple-choice knapsack problem. Eur J Oper Res **83**(2), 394–410 (1995)
21. Pissis, S.P.: MoTeX-II: structured MoTif eXtraction from large-scale datasets. BMC Bioinformatics **15**, 235 (2014)
22. Sinha, P., Zoltners, A.A.: The multiple-choice knapsack problem. Operations Research **27**(3), 431–627 (1979)

23. Theodorakopoulos, G., Shokri, R., Troncoso, C., Hubaux, J., Boudec, J.L.: Prolonging the hide-and-seek game: Optimal trajectory privacy for location-based services. In: WPES. pp. 73–82 (2014)
24. Verykios, V.S., Elmagarmid, A.K., Bertino, E., Saygin, Y., Dasseni, E.: Association rule hiding. TKDE **16**(4), 434–447 (2004)
25. Wang, D., He, Y., Rundensteiner, E., Naughton, J.F.: Utility-maximizing event stream suppression. In: SIGMOD. pp. 589–600 (2013)

A Omitted Proofs

Proof (Lemma 1).

C1: Index j in TFS-ALGO runs over the positions of string W; at any moment it indicates the ending position of the currently considered length-k substring of W. When $C[j-k+1]=1$ (Lines 9-11) TFS-ALGO never appends $W[j]$, *i.e.,* the last letter of a sensitive length-k substring, implying that, by construction of C, no $W[i\mathinner{.\,.}i+k-1]$ with $C[i]=1$ occurs in X.

P1: When $C[j-k]=C[j-k+1]=0$ (Lines 7-8) TFS-ALGO appends $W[j]$ to X, thus the order of $W[j-k\mathinner{.\,.}j-1]$ and $W[j-k+1\mathinner{.\,.}j]$ is clearly preserved. When $C[j-k]=0$ and $C[j-k+1]=1$, index f stores the starting position on W of the $(k-1)$-length suffix of the last non-sensitive substring appended to X (see also Fig. 1). **C1** ensures that no sensitive substring is added to X in this case, nor when $C[j-k]=C[j-k+1]=1$. The next letter will thus be appended to X when $C[j-k]=1$ and $C[j-k+1]=0$ (Lines 12-17). The condition on Line 13 is satisfied if and only if the last non-sensitive length-k substring appended to X overlaps with the immediately succeeding non-sensitive one by $k-1$ letters: in this case, the last letter of the latter is appended to X by Line 14, clearly maintaining the order of the two. Otherwise, Line 17 will append $W[j-k+1\mathinner{.\,.}j]$ to X, once again maintaining the length-k substrings' order. Conversely, by construction, any $U\in\Sigma^k$ occurs in X only if it equals a length-k non-sensitive substring of W. The only occasion when a letter from W is appended to X more then once is when Line 17 is executed: it is easy to see that in this case, because of the occurrence of #, each of the $k-1$ repeated letters creates exactly one $U\notin\Sigma^k$, without introducing any new length-k string over Σ nor increasing the occurrences of a previous one. Finally, Line 14 does not introduce any new $U\in\Sigma^k$ except for the one present in W, nor any extra occurrence of the latter, because it is only executed when two consecutive non-sensitive length-k substrings of W overlap exactly by $k-1$ letters.

P2: It follows from the proof for **C1** and **P1**.

P3: Letter # is added only by Line 16, which is executed only when $C[j-k]=1$ and $C[j-k+1]=0$. This can be the case up to $\lceil\frac{n-k+1}{2}\rceil$ times as array C can have alternate values only in the first $n-k+1$ positions. By construction, X cannot start with # (Lines 2-4), and thus the maximal number of occurrences of # is $\lfloor\frac{n-k+1}{2}\rfloor$. By construction, letter # in X is followed by at least k letters (Line 17): the leftmost non-sensitive substring following a sequence of one or more occurrences of sensitive substrings in W.

P4: *Upper bound.* TFS-ALGO increases the length of string X by more than one letter only when letter # is added to X (Line 16). Every time Lines 16-17 are executed, the length of X increases by $k+1$ letters. Thus the length of X is maximized when the maximal number of occurrences of # is attained. This length is thus bounded by $\lceil\frac{n-k+1}{2}\rceil \cdot k + \lfloor\frac{n-k+1}{2}\rfloor$.

Tightness. For the lower bound, let $W = a^n$ and a^k be sensitive. The condition at Line 3 is not satisfied because no element in C is set to 0: $j = n$. Then the condition on Line 5 is also not satisfied because $j = n$, and thus TFS-ALGO outputs the empty string. A *de Bruijn sequence* of order k over an alphabet Σ is a string in which every possible length-k string over Σ occurs exactly once as a substring. For the upper bound, let W be the order-$(k-1)$ de Bruijn sequence over alphabet Σ, k be even, and $S = \{1,3,5,\ldots,n-k\}$. $C[0] = 0$ and so Line 4 will add the first k letters of W to X. Then observe that $C[1] = 1, C[2] = 0; C[3] = 1, C[4] = 0, \ldots$, and so on; this sequence of values corresponds to satisfying Lines 12 and 9 alternately. Line 9 does not add any letter to X. The *if* statement on Line 13 will always fail because of the de Bruijn sequence property. We thus have a sequence of the non-sensitive length-k substrings of W interleaved by occurrences of # appended to X. TFS-ALGO thus outputs a string of length $\lceil\frac{n-k+1}{2}\rceil \cdot k + \lfloor\frac{n-k+1}{2}\rfloor$ (see Example 4).

*Example 4 (Illustration of **P3**).* Let $k = 4$. We construct the order-3 de Bruijn sequence $W = \texttt{baaabbbaba}$ of length $n = 10$ over alphabet $\Sigma = \{\texttt{a},\texttt{b}\}$, and choose $\mathcal{S} = \{1,3,5\}$. TFS-ALGO constructs:

$$X = \texttt{baaa\#aabb\#bbba\#baba}.$$

The upper bound of $\lceil\frac{n-k+1}{2}\rceil \cdot k + \lfloor\frac{n-k+1}{2}\rfloor = 19$ on the length of X is attained. □

Proof (Lemma 3). Consider the following renaming technique. Each length-ℓ substring of the collection is assigned a *lexicographic rank* from the range $\{1,\ldots,||\mathcal{B}||\}$. Each string in $\mathcal{B}$ is converted to a two-letter string as follows. The first letter is the lexicographic rank of its length-ℓ prefix and the second letter is the lexicographic rank of its length-ℓ suffix. We thus obtain a new *collection* $\mathcal{B}'$ of two-letter strings. Computing the ranks for all length-ℓ substrings in $\mathcal{B}$ can be implemented in $\mathcal{O}(||\mathcal{B}||)$ time by employing radixsort to sort Σ and then the well-known LCP data structure over the concatenation of strings in $\mathcal{B}$ [9]. The FO-SSM problem is thus solved by finding a shortest string containing every element of $\mathcal{B}'$ as a distinct substring. Since $\mathcal{B}'$ consists of two-letter strings only we can solve the problem in $\mathcal{O}(|\mathcal{B}'|)$ time by applying Lemma 2. The statement follows. □

Proof (Lemma 4). **C1** and **P2** hold trivially for Y as no length-k substring over Σ is added or removed from X. Let $X = X_1\#\ldots\#X_N$. The order of non-sensitive length-k substrings within X_i, for all $i \in [1,N]$, is preserved in Y. Thus for any p-chain $\mathcal{J}_W$ of $\mathcal{I}_W$, there is a p-chain $\mathcal{J}_Y$ of $\mathcal{I}_Y$ such that $\mathcal{J}_W \equiv \mathcal{J}_Y$ ($\Pi 1$ is preserved). **P3** also holds trivially for Y as no occurrence of # is added. Since $|Y| \le |X|$, for **P4**, it suffices to note that the construction of W in the proof of tightness in Lemma 1 (see also Example 4) ensures that there is no suffix-prefix overlap of length $k-1$ between *any* pair of length-k substrings of Y over Σ^* due to the property of the order-$(k-1)$ de Bruijn sequence. Thus the upper bound of $\lceil \frac{n-k+1}{2} \rceil \cdot k + \lfloor \frac{n-k+1}{2} \rfloor$ on the length of X is also tight for Y.

The minimality on the length of Y follows from the minimality of $|X|$ and the correctness of Lemma 3 that computes a shortest such string. □

Proof (Lemma 5). We reduce the NP-hard *Multiple Choice Knapsack* (MCK) problem [22] to $\text{MCSR}_{k=1}$ in polynomial time. In MCK, we are given a set of elements subdivided into δ, mutually exclusive classes, $C_1, \ldots, C_\delta$, and a knapsack. Each class C_i has $|C_i|$ elements. Each element $j \in C_i$ has an arbitrary cost $c_{ij} \geq 0$ and an arbitrary weight w_{ij}. The goal is to minimize the total cost (Eq. 1) by filling the knapsack with one element from each class (constraint II), such that the weights of the elements in the knapsack satisfy constraint I, where constant $b \geq 0$ represents the minimum allowable total weight of the elements in the knapsack:

$$\min \sum_{i \in [1,\delta]} \sum_{j \in C_i} c_{ij} \cdot x_{ij} \tag{1}$$

subject to the constraints: (I) $\sum_{i \in [1,\delta]} \sum_{j \in C_i} w_{ij} \cdot x_{ij} \geq b$, (II) $\sum_{j \in C_i} x_{ij} = 1, \quad i = 1, \ldots \delta$, and (III) $x_{ij} \in \{0,1\}, \quad i = 1, \ldots, \delta, \quad j \in C_i$.

The variable x_{ij} takes value 1 if the element j is chosen from class C_i, 0 otherwise (constraint III). We reduce any instance $\mathtt{I}_{\text{MCK}}$ to an instance $\mathtt{I}_{\text{MCSR}_{k=1}}$ in polynomial time, as follows:

(I) Alphabet Σ consists of letters α_{ij}, for each $j \in C_i$ and each class C_i, $i \in [1,\delta]$.

(II) We set $Y = \alpha_{11}\alpha_{12}\ldots\alpha_{1|C_1|}\#\ldots\#\alpha_{\delta 1}\alpha_{\delta 2}\ldots\alpha_{\delta|C_\delta|}\#$. Every element of Σ occurs exactly once: $\text{Freq}_Y(\alpha_{ij}) = 1$. Letter $\#$ occurs δ times in Y. For convenience, let us denote by $\mu(i)$ the ith occurrence of $\#$ in Y.

(III) We set $\tau = 2$ and $\theta = \delta - b$.

(IV) $\text{Ghost}(\mu(i), \alpha_{ij}) = c_{ij}$ and $\text{Sub}(\mu(i), \alpha_{ij}) = 1 - w_{ij}$. The functions are otherwise *not defined*.

This is clearly a polynomial-time reduction. We now prove the correspondence between a solution $S_{\mathtt{I}_{\text{MCK}}}$ to the given instance $\mathtt{I}_{\text{MCK}}$ and a solution $S_{\mathtt{I}_{\text{MCSR}_{k=1}}}$ to the instance $\mathtt{I}_{\text{MCSR}_{k=1}}$.

We first show that if $S_{\mathtt{I}_{\text{MCK}}}$ is a solution to $\mathtt{I}_{\text{MCK}}$, then $S_{\mathtt{I}_{\text{MCSR}_{k=1}}}$ is a solution to $\mathtt{I}_{\text{MCSR}_{k=1}}$. Since the elements in $S_{\mathtt{I}_{\text{MCK}}}$ have minimum $\sum_{i \in [1,\delta]} \sum_{j \in C_i} c_{ij} \cdot x_{ij}$, $\text{Freq}_Y(\alpha_{ij}) = 1$, and $\tau = 2$, the letters $\alpha_1, \ldots, \alpha_\delta$ corresponding to the selected elements lead to a Z that incurs a minimum

$$\sum_{i \in [1,\delta]} \sum_{\substack{j = \mu(i): \text{Freq}_Y(Z[j]) < \tau \\ \text{Freq}_Z(Z[j]) \geq \tau}} \text{Ghost}(j, Z[j]). \tag{2}$$

In addition, each letter $Z[j]$ that is considered by the inner sum of Eq. 2 corresponds to a single occurrence of $\#$, and these are all the occurrences of $\#$. Thus we obtain that

$$\sum_{i\in[1,\delta]}\sum_{\substack{j=\mu(i):\text{Freq}_Y(Z[j])<\tau\\ \text{Freq}_Z(Z[j])\geq\tau}}\text{Ghost}(j,Z[j])=\sum_{\substack{i:Y[i]=\#,\ \text{Freq}_Y(Z[i])<\tau\\ \text{Freq}_Z(Z[i])\geq\tau}}\text{Ghost}(i,Z[i]) \tag{3}$$

(*i.e.,* condition I in Problem 3 is satisfied). Since the elements in $S_{\mathtt{I}_{\text{MCK}}}$ have total weight $\sum_{i\in[1,\delta]}\sum_{j\in C_i} w_{ij}\cdot x_{ij}\geq b$, the letters $\alpha_1,\ldots,\alpha_\delta$ they map to lead to a Z with $\sum_{i\in[1,\delta]}\sum_{j\in C_i}(1-\text{Sub}(\mu(i),\alpha_i))\cdot x_{ij}\geq\delta-\theta$, which implies

$$\sum_{i\in[1,\delta]}\sum_{j\in C_i}\text{Sub}(\mu(i),\alpha_{ij})\cdot x_{ij}=\sum_{i:Y[i]=\#}\text{Sub}(i,Z[i])\leq\theta \tag{4}$$

(*i.e.,* condition II in Problem 3 is satisfied). $S_{\mathtt{I}_{\text{MCSR}_{k=1}}}$ is thus a solution to $\mathtt{I}_{\text{MCSR}_{k=1}}$.

We finally show that, if $S_{\mathtt{I}_{\text{MCSR}_{k=1}}}$ is a solution to $\mathtt{I}_{\text{MCSR}_{k=1}}$, then $S_{\mathtt{I}_{\text{MCK}}}$ is a solution to $\mathtt{I}_{\text{MCK}}$. Since each $\#_i$, $i\in[1,\delta]$, is replaced by a single letter α_i in $S_{\mathtt{I}_{\text{MCSR}_{k=1}}}$, exactly one element will be selected from each class C_i (*i.e.,* conditions II-III of MCK are satisfied). Since the letters in $S_{\mathtt{I}_{\text{MCSR}_{k=1}}}$ satisfy condition I of Problem 3, every element of Σ occurs exactly once in Y, and $\tau=2$, their corresponding selected elements $j_1\in C_1,\ldots,j_\delta\in C_\delta$ will have a minimum total cost. Since $S_{\mathtt{I}_{\text{MCSR}_{k=1}}}$ satisfies $\sum_{i:Y[i]=\#}\text{Sub}(i,Z[i])=\sum_{i\in[1,\delta]}\sum_{j\in C_i}\text{Sub}(\mu(i),\alpha_{ij})\cdot x_{ij}\leq\theta$, the selected elements $j_1\in C_1,\ldots,j_\delta\in C_\delta$ that correspond to $\alpha_1\ldots,\alpha_\delta$ will satisfy $\sum_{i\in[1,\delta]}\sum_{j\in C_i}(1-w_{ij})\cdot x_{ij}\leq\delta-b$, which implies $\sum_{i\in[1,\delta]}\sum_{j\in C_i}w_{ij}\cdot x_{ij}\geq b$ (*i.e.,* condition I of MCK is satisfied). Therefore, $S_{\mathtt{I}_{\text{MCK}}}$ is a solution to $\mathtt{I}_{\text{MCK}}$. The statement follows. □

Proof (Lemma 6). It should be clear that if we conceptually extend Σ with the empty string, our approach takes into account the possibility of deleting (instead of replacing) an occurrence of #. To ease comprehension though we only describe the case of letter replacements.

Step 1: Given Y, Σ, k, δ, and τ, we construct a set $\mathcal{C}$ of *candidate τ-ghosts* as follows. The candidates are at most $(|Y|-k+1-k\delta)+(k\delta\sigma)=\mathcal{O}(|Y|+k\sigma\delta)$ distinct strings of length k. The first term corresponds to all substrings of length k over Σ occurring in Y (*i.e.,* if Y did not contain #, we would have $|Y|-k+1$ such substrings; each of the δ # causes the loss of k such substrings). The second term corresponds to all possible substrings of length k that may be introduced in Z but do not occur in Y. For any string U from the set of these $\mathcal{O}(|Y|+k\delta\sigma)$ strings, we want to compute $\text{Freq}_Y(U)$ and its *maximal frequency* in Z, denoted by $\max\text{Freq}_Z(U)$, *i.e.,* the largest possible frequency that U can have in Z, to construct set $\mathcal{C}$. Let us denote by S_{ij} the string of length $2k-1$, containing the k consecutive length-k substrings, obtained after replacing the ith occurrence of # with letter j in Y.

(I) If $\text{Freq}_Y(U)\geq\tau$, U by definition can never become τ-ghost in Z, and we thus exclude it from $\mathcal{C}$. $\text{Freq}_Y(U)$, for all U occurring in Y, can be computed in $\mathcal{O}(|Y|)$ total time using the suffix tree of Y.

(II) If $\max \text{Freq}_Z(U) < \tau$, U by definition can never become τ-ghost in Z, and we thus exclude it from $\mathcal{C}$. $\max \text{Freq}_Z(U)$ can be computed by adding to $\text{Freq}_Y(U)$, the maximum additional number of occurrences of U caused by a letter replacement among all possible letter replacements. We sum up this quantity for each U and for all replacements of occurrences of # to obtain $\max \text{Freq}_Z(U)$. To do this, we first build the generalized suffix tree of $Y, S_{11}, \ldots, S_{\delta\sigma}$ in $\mathcal{O}(|Y| + k\delta\sigma)$ time. We then spell $S_{i1}, \ldots, S_{i\sigma}$, for all i, in the generalized suffix tree in $\mathcal{O}(k\sigma)$ time per i. We exploit suffix links to spell the length-k substrings of S_{ij} in $\mathcal{O}(k)$ time and memorize the maximum number of occurrences of U caused by replacing the ith occurrence of # among all j. We represent set $\mathcal{C}$ on the generalized suffix tree by marking the corresponding nodes, and we denote this representation by $T(\mathcal{C})$. The total size of this representation is $\mathcal{O}(|Y| + k\sigma\delta)$.

Step 2: We now want to construct an instance of the MCK problem using $T(\mathcal{C})$. We first set letter j as element α_{ij} of class C_i. We then set c_{ij} equal to the sum of the Ghost function cost incurred by replacing the ith occurrence of # by letter j for all (at most k) affected length-k substrings that are marked in $T(\mathcal{C})$. The main assumption of our heuristic is precisely the fact that we assume that this letter replacement will force all of these affected length-k substrings becoming τ-ghosts in Z. The computation of c_{ij} is done as follows. For each (i, j), $i \in [1, \delta]$ and $j \in [1, \sigma]$, we have k substrings whose frequency changes, each of length k. Let U be one such pattern occurring at position t of Z, where $\mu(i) - k + 1 \leq t \leq \mu(i)$ and $\mu(i)$ is the ith occurrence of # in Y. We check if U is marked in $T(\mathcal{C})$ or not. If U is not marked we add nothing to c_{ij}. If U is marked, we increment c_{ij} by $\text{Ghost}(t, U)$. We also set $w_{ij} = \text{Sub}(i, j)$ (as stated above, any letter that reinstates a sensitive pattern is assigned a weight $\text{Sub} > \theta$, so that it cannot be selected to replace an occurrence of # in Step 3). Similar to Step 1, the total time required for this computation is $\mathcal{O}(|Y| + k\sigma\delta)$.

Step 3: In Step 2, we have computed c_{ij} and w_{ij}, for all i, j, $i \in [1, \delta]$ and $j \in [1, \sigma]$. We thus have an instance of the MCK problem. We solve it and translate the solution back to a (suboptimal) solution of the MCSR problem: the element α_{ij} chosen by the MCK algorithm from class C_i corresponds to letter j and it is used to replace the ith occurrence of #, for all $i \in [1, \delta]$. The cost of solving MCK depends on the chosen algorithm and is given by a function $\mathcal{T}(\delta, \sigma)$.

Thus, the total cost of MCSR-ALGO is $\mathcal{O}(|Y| + k\delta\sigma + \mathcal{T}(\delta, \sigma))$. □

B Additional Details on Anonymization

Anonymization is a direction in privacy-preserving data mining which is applied to individual-specific data and aims to prevent the disclosure of individuals' identity and/or information that individuals are not willing to be associated with [3,18,10]. On the other hand, our approach is applied to a string modeling information that does not necessarily refer to specific individuals and aims to protect sensitive patterns that model confidential knowledge rather than values individuals do not want to be associated with. For example, our approach may be applied to a string comprised of letters corresponding to orders of different products by a business. In this case, subsequences of ordered products that provide competitive advantage to the business [14] are treated as sensitive patterns and should be concealed from the disseminated string. The fact that anonymization methods deal with individual-specific data and aim to prevent privacy threats other than confidential knowledge exposure leads to fundamentally different protection principles and methods than ours. Thus, our work is related to anonymization approaches in that it shares the general objective of protecting string data with [3,2] and that of protecting data while supporting string mining with the works of [5] and [7]. However, our work considers different input data and has a fundamentally different privacy objective than [3,2,5,7]. Specifically, these works consider a collection of strings instead of a long string and employ privacy objectives which do not aim to reduce the frequency of sensitive length-k substrings to zero. Therefore, they cannot be applied to address our problem.