Combinatorial Algorithms for String Sanitization

GIULIA BERNARDINI, University of Milano - Bicocca and CWI
HUIPING CHEN, King's College London
ALESSIO CONTE, University of Pisa
ROBERTO GROSSI, University of Pisa and ERABLE Team
GRIGORIOS LOUKIDES, King's College London
NADIA PISANTI, University of Pisa and ERABLE Team
SOLON P. PISSIS, CWI, Vrije Universiteit Amsterdam, and ERABLE Team
GIOVANNA ROSONE, University of Pisa
MICHELLE SWEERING, CWI

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 article, we consider the problem of sanitizing a string by concealing the occurrences of sensitive patterns, while maintaining data utility, in two settings that are relevant to many common string processing tasks.

In the first setting, we aim to generate the minimal-length string that preserves the order of appearance and frequency of all non-sensitive patterns. Such a string allows accurately performing tasks based on the sequential nature and pattern frequencies of the string. To construct such a string, we propose a time-optimal algorithm, TFS-ALGO. We also propose another 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 are constructed by concatenating non-sensitive parts of the input string. However, it is possible to detect the sensitive patterns by "reversing" the concatenation operations. In response, we propose a heuristic, MCSR-ALGO, which replaces letters in the strings output by the algorithms with carefully selected letters, so that sensitive patterns are not reinstated, implausible patterns are not introduced, and occurrences of spurious patterns are prevented. In the second setting, we aim to generate a string that is at minimal edit distance from the original string, in addition to preserving the order of appearance and frequency of all non-sensitive patterns. To construct such a string,

HC is supported by a CSC scholarship. NP, AC, RG, and GR are partially supported by the Università di Pisa under the "PRA – Progetti di Ricerca di Ateneo" (Institutional Research Grants) - Project no. PRA_2020-2021_26 "Metodi Informatici Integrati per la Biomedica." AC and RG are partially supported by the Italian Ministry of University and Research (MIUR) PRIN Project no. 20174LF3T8 "AHeAD." MS is supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003. GR and NP are partially supported by MIUR-SIR project CMACBioSeq grant n. RBSI146R5L.

Authors' addresses: G. Bernardini, Dip. di Informatica, Sistemistica e Comunicazione, Viale Sarca, 336, I-20126 Milano, Italy; email: giulia.bernardini@unimib.it; H. Chen and G. Loukides, Department of Informatics, King's College London, Bush House, 30 Aldwych, London, WC2B 4BG, UK; emails: {huiping.chen, grigorios.loukides}@kcl.ac.uk; A. Conte, R. Grossi, N. Pisanti, and G. Rosone, Informatica, Università di Pisa, Largo Pontecorvo 3, 56127 Pisa, Italy; emails: {alessio.conte, giovanna.rosone, roberto.grossi, nadia.pisanti}@unipi.it; S. P. Pissis and M. Sweering, CWI, P.O. Box 94079, 1090 GB Amsterdam, NETHERLANDS; email: {solon.pissis, michelle.sweering}@cwi.nl.

1556-4681/2020/12-ART8 $15.00
https://doi.org/10.1145/3418683

we propose an algorithm, ETFS-ALGO, based on solving specific instances of approximate regular expression matching.

We implemented our sanitization approach that applies TFS-ALGO, PFS-ALGO, and then MCSR-ALGO, and experimentally show that it is effective and efficient. We also show that TFS-ALGO is nearly as effective at minimizing the edit distance as ETFS-ALGO, while being substantially more efficient than ETFS-ALGO.

CCS Concepts: • **Security and privacy** → **Data anonymization and sanitization**; • **Mathematics of computing** → *Combinatorics on words*;

Additional Key Words and Phrases: Data privacy, data sanitization, knowledge hiding, sequences, strings, sensitive knowledge

ACM Reference format:

Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giovanna Rosone, and Michelle Sweering. 2020. Combinatorial Algorithms for String Sanitization. *ACM Trans. Knowl. Discov. Data* 15, 1, Article 8 (December 2020), 34 pages.
https://doi.org/10.1145/3418683

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 deoxyribonucleic acid (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 [30], and retailers outsource their data to marketing agencies who perform tasks such as mining frequent patterns from the strings [31].

However, disseminating a string intact may result in the exposure of confidential knowledge, such as trips to mental health clinics in transportation data [48], query terms revealing political beliefs or sexual orientation of individuals in web data [38], or diseases associated with certain parts of DNA data [34]. 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.

1.1 Our Model and Settings

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 by applying a sequence of edit operations. The computational task is to determine this sequence of edit operations so that the transformed string satisfies the desirable properties subject to the constraints. Clearly, the constraints and the properties must be specified based on the application.

Under the CSD model, we consider two specific settings addressing practical considerations in common string processing applications: the *Minimal String Length* (MSL) setting, in which the goal is to produce a shortest string that satisfies the set of constraints and the set of desirable properties, and the *Minimal Edit Distance* (MED) setting, in which the goal is to produce a string that satisfies the set of constraints and the set of desirable properties and is at MED from W. In the following, we discuss each setting in more detail.

MSL *Setting*. In this setting, 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 Total order, Frequency, Sanitization (TFS). Note that it is straightforward to hide substrings of *arbitrary* lengths 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.

The MSL 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, 13, 25, 31]. 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 [26], to be performed accurately. By satisfying **P2**, X allows computing the frequency of length-k substrings and hence mining frequent length-k substrings [41] 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.

MED *Setting*. In this setting, the sanitized version X_{ED} of string W must satisfy the properties **P1** and **P2**, subject to the constraint **C1**, and also be at MED from string W. Constructing such a string X_{ED} allows many tasks that are based on edit distance to be performed accurately. Examples of such tasks are frequent pattern mining [44], clustering [28], entity extraction [51] and range query answering [33], which are important in domains such as bioinformatics [44], text mining [51], and speech recognition [20].

Note, existing works for sequential data sanitization (e.g., [13, 25, 27, 31, 50]) or anonymization (e.g., [4, 14, 17]) cannot be applied to our settings (see Section 2 for details).

1.2 Our Contributions

We define the TFS problem for string sanitization and a variant of it, referred to as Partial order, Frequency, Sanitization (PFS), which aims at producing an even shorter string Y by relaxing **P1** of TFS. We also develop algorithms for TFS and PFS. Our algorithms construct strings X and Y using a separator letter #, which is not contained in the alphabet of W, ensuring that sensitive patterns do not occur 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 Minimum-Cost Separators Replacement (MCSR) 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. In addition, we define the problem of constructing X_{ED} in the MED setting, which is referred to as Edit-distance, Total order, Frequency, Sanitization (ETFS), and design an algorithm framework to solve it.

Our work makes the following specific contributions:

1. We design an algorithm, TFS-ALGO, for solving the TFS problem in $O(kn)$ time, where n is the length of W. In fact, we prove that $O(kn)$ time is worst-case optimal by showing that the length of X is in $\Theta(kn)$ in the worst case. The output of TFS-ALGO is a string X consisting of a sequence of substrings over the alphabet of W separated by # (see Example 1.1 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 $O(n)$-sized representation of X with respect to W in $O(n)$ time. See Section 4.

Example 1.1. Let W = aabaaacbcbbbaabbacaab, $k = 4$, and the set of sensitive patterns be {baaa, bbaa}. The string X = aabaa#aaacbcbbba#baabbacaab consists of three substrings over the alphabet {a, b, c} separated by #. Note that no sensitive pattern occurs in X, while all non-sensitive substrings of length $k = 4$ have the same frequency in W and in X (e.g., aaba appears once), and they appear in the same order in W and in X (e.g., aaba precedes abaa). 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 caab 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* (**Π1**) that preserves the order of appearance only for sequences of consecutive non-sensitive length-k substrings that overlap by $k - 1$ letters. In other words, **Π1** requires preserving the order of appearance of any two non-sensitive length-k substrings U, V for which the following two conditions hold: (I) U and V occur consecutively in W, and (II) the length-$(k - 1)$ suffix of U is the same as the length-$(k - 1)$ prefix of V. This makes sense because the order of two consecutive non-sensitive length-k substrings with no length-$(k - 1)$ overlap has anyway been "interrupted" (by one or more sensitive patterns). We exploit this observation to shorten the string further. Specifically, we design an algorithm that solves PFS in the optimal $O(n + |Y|)$ time, where $|Y|$ is the length of Y, using the $O(n)$-sized representation of X. See Section 5.

Example 1.2 (Cont'd from Example 1.1). Recall that W = aabaaacbcbbbaabbacaab. A string Y is aaacbcbbba#aabaabbacaab. The order of aaba and abaa is preserved in Y as they are consecutive, non-sensitive, and the length-3 suffix of aaba is the same as the length-3 prefix of abaa (i.e., they have an overlap of $k-1=3$ letters). The order of abaa and aaac, which are consecutive 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, at a given support threshold τ, are prevented; and 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. We also show how to apply the heuristic, so that letter replacements do not result in *implausible* patterns that may reveal the location of sensitive patterns. An implausible pattern is a string which is unlikely to occur in Z as a substring. For example, such a pattern may correspond to an impossible or unlikely trip in a sanitized movement dataset Z. When an occurrence of an implausible pattern is identified in Z, it becomes easier to identify the letter that replaced a # in the implausible pattern, and thus recover the sensitive pattern. To prevent this, we first define an implausible pattern as a statistically unexpected string. Our definition is based on a statistical significance measure computed over a reference dataset [7, 15, 42]. Specifically, an implausible pattern is a substring whose frequency in W is significantly smaller than its expected frequency in W. Then, we modify MCSR-ALGO, so that it does not replace any occurrence of # with letters that create implausible patterns. See Section 6.

Example 1.3 (Cont'd from Example 1.2). Recall that Y = aaacbcbbba#aabaabbacaab. Let $\tau = 1$. A string Z = aaacbcbbbacaabaabbacaab is produced by replacing letter # with letter c. Note that Z contains no sensitive pattern, nor a non-sensitive pattern of length-4 substring that could not be mined from W at a support threshold τ (i.e., a pattern that does not occur in W). In addition, Z contains no implausible pattern, such as bbab, which is not expected to occur in W, according to an established statistical significance measure for strings [7, 15, 42].

4. We design an algorithm for solving the ETFS problem. The algorithm, called ETFS-ALGO, is based on a connection between ETFS and the approximate regular expression matching problem [37]. Given a string W and a regular expression E, the latter problem seeks to find a string T that matches E and is at MED from W. ETFS-ALGO solves the ETFS problem in $O(k|\Sigma|n^2)$ time, where $|\Sigma|$ is the size of the alphabet of W. See Section 7.

Example 1.4. Let W = aaaaaab, $k = 4$, and the set of sensitive patterns be {aaaa, aaab}. TFS-ALGO constructs string $X = \varepsilon$, where ε is the empty string, with $d_E(W, X) = 7$. On the contrary, ETFS-ALGO constructs string X_{ED} = aaa#aab with $d_E(W, X_{\text{ED}}) = 1 < 7$. Clearly, string X_{ED} is more suitable for applications, which are based on measuring sequence similarity.

5. For the MSL setting, we implemented our combinatorial approach for sanitizing a string W (i.e., the aforementioned algorithms implementing the pipeline $W \to X \to Y \to Z$) and show its effectiveness and efficiency on real and synthetic data. We also show that it possible to produce a string Z that does not contain implausible patterns, while incurring insignificant additional utility loss. See Section 8.

6. For the MED setting, we implemented ETFS-ALGO and experimentally compared it with TFS-ALGO. Interestingly, we demonstrate that TFS-ALGO constructs optimal or near-optimal solutions to the ETFS problem in practice. This is particularly encouraging because TFS-ALGO is linear in the length of the input string n, whereas ETFS-ALGO is quadratic in n. See Section 8.

A preliminary version of this article, without the method that avoids implausible patterns and without contributions 4 and 6, appeared in [10]. Furthermore, we include here all proofs omitted from [10], as well as additional examples and discussion of related work.

2 RELATED WORK

We review related work in data sanitization (*a.k.a. knowledge hiding*) and data anonymization, two of the main topics in the area of privacy-preserving data mining [6, 12]. Data sanitization aims at concealing confidential knowledge, so that it is not easily discovered by mining a disseminated dataset [1, 25, 49]. For example, data sanitization may be used by a business to prevent a recipient of a dataset from inferring that a specific set of products (e.g., baking powder and flour) is purchased by many customers of the business [49]. This set of products needs to be concealed, as it provides competitive advantage to the business.

On the other hand, data anonymization [4, 21, 36] aims at preventing a data recipient from inferring information about individuals whose information is contained in the input dataset [22]. This includes inferences about the identity of an individual (identity disclosure), about whether or not an individual's information is contained in the output dataset (membership disclosure), as well as inferences that generally depend on an individual's information (inferential disclosure). For example, data anonymization works are used to prevent a data recipient from inferring the identity of an individual based on the products purchased by the individual, or from inferring that the individual has purchased a sensitive product (e.g., a medicine revealing their health condition) [52].

2.1 Data Sanitization

Existing data sanitization approaches can be classified, based on the type of data they are applied to, into those applied to a collection of records and others applied to a single sequence.

We first discuss data sanitization approaches that are applied to a collection of records. A record can be a set of values (itemset) [39, 46, 49], a trajectory [1], or a sequence [1, 25, 27]. In set-valued (transaction) datasets, the confidential knowledge to be hidden is typically modeled as a set of itemsets [46], association rules [49], or classification rules [39]. In trajectory datasets, the confidential knowledge is modeled as a set of subtrajectories [1]. Last, in sequential datasets, the confidential knowledge is modeled as a set of sequential patterns occurring in the dataset [1, 25, 27].

In what follows, we review three data sanitization approaches [1, 25, 27], which are applied to a collection of sequences, since they are the most relevant to our work. The key difference of these approaches from our work is that they aim to hide sensitive patterns occurring as *subsequences* (not only as substrings) in the input *collection* (not in a single, long string). Moreover, they aim to hide sensitive patterns when these are *sufficiently frequent*; i.e., when a sensitive pattern occurs as a subsequence of least τ records, where τ is a given minimum frequency threshold. The hiding of a sensitive pattern is then performed by modifying some of the records in the collection (e.g., by letter deletion [1]), so that fewer than τ records contain the sensitive pattern as a subsequence. In our work, **C1** implies that no occurrence of a sensitive pattern exists in the sanitized sequence.

The problem of sanitizing a collection of sequences was first proposed by Abul et al. [1]. The authors developed a heuristic that applies deletion of letters contained in sensitive patterns. The heuristic aims to minimize the number of deleted letters in the collection. However, it does not focus on minimizing changes to the set of non-sensitive frequent sequential patterns that are incurred by deletions. In response, Gkoulalas-Divanis et al. [25] developed a heuristic that avoids such changes, hence improving data utility for frequent sequential pattern mining and tasks based on it. The heuristic of [25] first selects a sufficiently large subset of records to sanitize, favoring records that can be sanitized with few deletions. Then, it sanitizes each selected record by constructing a graph that represents the matchings between the record and sensitive patterns, and searching for graph nodes corresponding to good letters to delete. However, due to the fact that graph search is computationally inefficient, the heuristic searches only a small part of the graph.

Gwadera et al. [27] proposed a heuristic, called Permutation Hiding (PH). PH addresses the limitation of [1], as it aims to minimize changes to the set of non-sensitive frequent sequential patterns. Also, it addresses the limitation of [25], as it avoids the expensive graph search. Furthermore, PH employs both letter permutation and deletion to hide sensitive patterns. Permuting the letters of a sensitive pattern hides the pattern but may change the set of non-sensitive frequent sequential patterns. Thus, PH explores the space of possible permutations of the letters of a sensitive pattern to find a permutation that minimizes the number of such changes. When this is not possible, PH resorts to letter deletion.

Table 1. The τ-lost and τ-ghost Patterns, for $\tau = 1$, Created by Applying the PH Heuristic [27] and Our Method on the String of Example 2.1

	τ-lost	τ-ghost
PH [27]	{abaa, aaac, aacb, bbba, baab, aabb, abba, bbac, baca, acaa, caab}	{abac, bacb, bbbb, bbbc, bbca, bcab}
Our method	$\emptyset$	$\emptyset$

Thus, in summary, our approach differs from existing approaches that are applied to a collection of sequences [1, 25, 27], in terms of: (I) input dataset (a collection of strings vs. a single string); (II) occurrences of a sensitive pattern that must be hidden (occurrences as a subsequence vs. occurrences as a substring); (III) data modification strategy (deletion and/or permutation vs. copying of non-sensitive substrings and letter replacement); and (IV) utility considerations (no guarantees on minimizing changes to non-confidential frequent sequential patterns vs. guarantees on utility properties). Although these data sanitization methods were designed for the general case of a collection of sequences, they could in principle be applied to a single string. Through the following example, we illustrate this point and also highlight the difference with respect to the goals of our methods.

Example 2.1. Let $W =$ aabaaacbcbbbaa$\underline{\text{bb}}$ac$\underline{\text{aa}}$b, $k = 4$, and the set of sensitive patterns be {baaa, bbaa}. Consider applying the PH heuristic [27] using a minimum frequency threshold $\tau = 1$. PH constructs a string $I =$ aaba**cbcbbb**bb*ca*b, deleting six letters of W that are represented by the special letter $\star$ for the sake of clarity. PH also creates non-sensitive length-k substrings that can be mined from W but cannot be mined from Z at frequency threshold τ, as well as non-sensitive length-k substrings that cannot be mined from W but can be mined from Z at frequency threshold τ. These substrings are referred to as *τ-lost* and *τ-ghost* patterns, respectively. Specifically, as shown in Table 1, PH created 11 τ-lost and 6 τ-ghost patterns. On the other hand, applying our approach (i.e., the pipeline TFS-ALGO→ PFS-ALGO→ MCSR-ALGO) with $\tau = 1$ produces a string $Z =$ aaacbcbbbacaabaabbacaab with neither τ-lost nor τ-ghost patterns, as mentioned in Example 1.3. The reader can perhaps share the intuition that string Z is more useful than string I, as Z preserves the set of non-sensitive frequent sequential patterns that can be mined at $\tau = 1$.

The main reason PH incurs substantially more τ-lost and τ-ghost patterns than our method is because it hides the sensitive patterns when they occur as subsequences of the input string. That is, it hides all occurrences of each sensitive pattern in the string, albeit only occurrences comprised of consecutive letters (i.e., substrings) need to be hidden in our setting. For instance, two occurrences of the letter a have been deleted from the suffix bbacaab of W to prevent the sensitive pattern bbaa from occurring as a subsequence (the subsequence is comprised of the underlined letters in W). Note, however, that pattern bbaa does not occur as a substring in this suffix of W.

In what follows, we review three data sanitization approaches [14, 31, 50], which are applied to a single sequence.

The work of Loukides et al. [31] is applied to a single event-sequence, in which each event is a multi-set of letters associated with a timestamp. Their work aims to hide sensitive patterns comprised of *a single letter*. Each such pattern is considered hidden when its relative frequency in any prefix of the event-sequence is sufficiently low. The hiding is performed by a dynamic-programming algorithm that applies letter deletion, while preserving the distribution of events across the sequence. The approach of [31] cannot be readily extended to hide sensitive patterns of length $k > 1$, which is our privacy objective. Moreover, it has a different utility criterion than our work, and it does not guarantee the satisfaction of the utility properties we consider here.

The work of Bonomi et al. [13] is applied to a single sequence and aims to prevent an attacker, who has background knowledge about the frequency distribution of sensitive patterns in the input sequence, from gaining additional knowledge about the frequency distribution of sensitive patterns by observing the sanitized sequence. This is performed by limiting the mutual information between the frequency distribution of sensitive patterns in the original and sanitized sequence. In other words, sensitive patterns are protected when their frequencies are similar in the input and in the sanitized sequence. On the other hand, in our work, we consider a setting where sensitive patterns are unknown to the attacker and aim to prevent the attacker from observing their presence in the sanitized sequence. The hiding of sensitive patterns in [13] is performed by heuristics which aim to apply a small amount of generalization [43]. Generalization replaces a letter with an aggregate letter that is not part of the sequence alphabet, thereby introducing uncertainty. Thus, the work of [13] aims to produce sanitized data with a low level of uncertainty and does not focus on guaranteeing the accuracy of mining frequent substrings comprised of the letters of the alphabet.

The work of Wang et al. [50] is applied to an event-sequence, in which each event is a single letter associated with a timestamp. Their work considers the problem of deleting events in a given sequence, so as to reduce the ability of an attacker to detect sensitive patterns, while maximizing the detection of non-sensitive patterns. A pattern is detected when it occurs as a subsequence within a specified time window of the sequence. To solve this problem, the approach of [50] deletes events from the sequence in order to maximize a weighted utility function expressed as a sum of terms. An occurrence of a non-sensitive (respectively, sensitive) pattern in the sequence contributes a positive (respectively, negative) term to this function. Thus, [50] considers protecting sensitive patterns that occur as subsequences rather than as substrings, and it aims to achieve a good balance between matching non-sensitive patterns and preventing the matching of sensitive patterns.

2.2 Data Anonymization

Data anonymization is a different direction in privacy-preserving data mining than data sanitization [2, 5]. Data anonymization has been the focus of many research works (see [5, 23] for surveys). This includes works for anonymizing string data [3, 4, 14, 17]. The works of Aggarwal and Yu [3, 4] aim to enforce k-anonymity [43] on a collection of strings. This is performed by first grouping strings, so that each group contains at least k similar strings, and then replacing the strings in each group with a carefully constructed synthetic string. The work of [14] aims to release differentially private [21] top-k frequent substrings from a collection of strings, where k denotes the number of frequent substrings required. This is performed by building a noisy summary data structure that represents the collection and then mining the top-k frequent substrings from the data structure. The work of [17] aims to release a differentially private collection of strings. This is performed by exploiting the variable-length n-gram model [35] and calibrating the noise needed to enforce differential privacy based on the model.

The aforementioned anonymization methods aim to prevent privacy threats other than eliminating sensitive substrings from a string to prevent their mining. The threats they are dealing with, following the terminology of [22], are: identity disclosure for [3, 4] and membership as well as inferential disclosure for [14, 17]. Thus, our work is related to anonymization approaches in that it shares the general objective of protecting string data with [3, 4] and that of protecting data while supporting string mining with the work of [14].

3 PRELIMINARIES, PROBLEM STATEMENTS, AND MAIN RESULTS

In this section, we start with providing some preliminary definitions. Then, we define our problems and introduce our main results. A summary of the acronyms introduced in the article is in Table 2.

Table 2. Acronyms Used Throughout

Acronym	Meaning
CSD	Combinatorial String Dissemination model
MSL	Minimal String Length setting
MED	Minimal Edit Distance setting
TFS	Total order, Frequency, Sanitization problem
PFS	Partial order, Frequency, Sanitization problem
MCSR	Minimum-Cost Separators Replacement problem
ETFS	Edit-distance, Total order, Frequency, Sanitization problem
PH	Permutation Hiding heuristic [27]
MCK	Multiple Choice Knapsack problem [29]
FO-SSM	Fixed-Overlap Shortest String with Multiplicities problem
SCS	Shortest Common Superstring problem [24]
OLD	Oldenburg dataset [1]
TRU	Trucks dataset [25]
MSN	MSNBC dataset [27]
DNA	The complete genome of *Escherichia coli* dataset [31]
SYN	Synthetic dataset

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 $T[i\,..\,j] = T[i] \dots T[j]$ 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 $T[0\,..\,j]$, and a *suffix* of T is a substring of the form $T[i\,..\,n-1]$. A *proper* prefix (suffix) of a string is not equal to the string itself. By $\text{Freq}_V(U)$ 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., $U[|U| - \ell\,..\,|U| - 1] = V[0\,..\,\ell - 1]$.

We fix a string W of length n over an alphabet $\Sigma = \{1, \dots, n^{O(1)}\}$ 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 $W[j\,..\,j+k-1] = W[i\,..\,i+k-1]$, 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 $W[i\,..\,i+k-1]$ 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 $\mathcal{I} = \{0, \dots, n-k\} \setminus \mathcal{S}$ 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 in U. (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* $U[i\,..\,i+k-1]$ has a suffix–prefix overlap of length $k-1$ with $U[j\,..\,j+k-1]$. 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 $U[u\,..\,u+k-1] = V[v\,..\,v+k-1]$, where u is the jth smallest element of $\mathcal{J}_U$ and v is the jth smallest of $\mathcal{J}_V$, for all $j \le |\mathcal{J}_U|$.

Given two strings U and V the *edit distance* $d_E(U, V)$ is defined as the minimum number of elementary edit operations (letter insertion, deletion, or substitution) to transform U to V.

The set of *regular expressions* over an alphabet Σ is defined recursively as follows [37]: (I) $a \in \Sigma \cup \{\varepsilon\}$, where ε denotes the empty string, is a regular expression. (II) If E and F are regular expressions, then so are EF, $E|F$, and E^*, where EF denotes the set of strings obtained by concatenating a string in E and a string in F, $E|F$ is the union of the strings in E and F, and E^* consists of all strings obtained by concatenating zero or more strings from E. Parentheses are used to override the natural precedence of the operators, which places the operator * highest, the concatenation next, and the operator | last. We state that a string T *matches* a regular expression E, if T is equal to one of the strings in E.

Problem Statements and Main Results. We define the following problem for the MSL setting.

Problem 1 (TFS). *Given W, k, $\mathcal{S}$, and $\mathcal{I}_W$ 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 *$Freq_X(U) = Freq_W(U)$, for all $U \in \Sigma^k \setminus \{W[i \,..\, i+k-1] : i \in \mathcal{S}\}$.*

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 3.1. *Let W be a string of length n over $\Sigma = \{1, \ldots, n^{O(1)}\}$. Given $k < n$ and $\mathcal{S}$,* TFS-ALGO *solves Problem 1 in $O(kn)$ time, which is worst-case optimal. An $O(n)$-sized representation of X can be built in $O(n)$ 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 consecutive non-sensitive length-k substrings that also overlap by $k-1$ letters (p-chains). This leads to the following problem for the MSL setting.

Problem 2 (PFS). *Given W, k, $\mathcal{S}$, and $\mathcal{I}_W$ construct a* shortest *string Y:*

C1 *Y does not contain any sensitive pattern.*
Π1 *There exists an injective function f from the p-chains of $\mathcal{I}_W$ to the p-chains of $\mathcal{I}_Y$ such that $f(\mathcal{J}_W) \equiv \mathcal{J}_W$ for any p-chain $\mathcal{J}_W$ of $\mathcal{I}_W$.*
P2 *$Freq_Y(U) = Freq_W(U)$, for all $U \in \Sigma^k \setminus \{W[i \,..\, i+k-1] : i \in \mathcal{S}\}$.*

Our second result, which builds on Theorem 3.1, is the following.

Theorem 3.2. *Let W be a string of length n over $\Sigma = \{1, \ldots, n^{O(1)}\}$. Given $k < n$ and $\mathcal{S}$,* PFS-ALGO *solves Problem 2 in the optimal $O(n + |Y|)$ time.*

To arrive at Theorems 3.1 and 3.2, 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 sensitive patterns or create implausible patterns. Given an integer threshold $\tau > 0$, we call a pattern $U \in \Sigma^k$ a $\tau - ghost$ 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 [40].

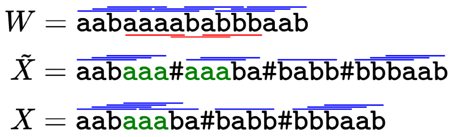

W = aabaaaababbbaab
$\tilde{X}$ = aabaaa#aaaba#babb#bbbaab
X = aabaaaba#babb#bbbaab

Fig. 1. Sensitive patterns are underlined in red; non-sensitive patterns are overlined 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.

Theorem 3.3. *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.

We define the following problem for the MED setting.

Problem 3 (ETFS). *Given W, k, $\mathcal{S}$, and $\mathcal{I}$, construct a string X_{ED} which is at* MED *from W and satisfies the following:*

C1 *X_{ED} does not contain any sensitive pattern.*
P1 *$\mathcal{I}_W \equiv \mathcal{I}_{X_{ED}}$, i.e., the t-chains $\mathcal{I}_W$ and $\mathcal{I}_{X_{ED}}$ are equivalent.*
P2 *$Freq_{X_{ED}}(U) = Freq_W(U)$, for all $U \in \Sigma^k \setminus \{W[i\mathinner{.\,.} i+k-1] : i \in \mathcal{S}\}$.*

We show how to reduce any instance of the ETFS problem to some instance of the approximate regular expression matching problem. In particular, the latter instance consists of a string of length n (string W) and a regular expression E of length $\mathcal{O}(k|\Sigma|n)$. We thus prove the claim of Theorem 3.4 by employing the $\mathcal{O}(|W| \cdot |E|)$-time algorithm of [37].

Theorem 3.4. *Let W be a string of length n over an alphabet Σ. Given $k < n$ and $\mathcal{S}$,* ETFS-ALGO *solves Problem 3 in $\mathcal{O}(k|\Sigma|n^2)$ time.*

4 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, we append the succeeding non-sensitive substring (in the t-predecessor order) 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 Figure 1).

R1 prevents U from occurring in X, and **R2** reduces the length of X (i.e., allows to hide 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 4.1 for details of the workings of TFS-ALGO and Figure 1 for an example). We next show that string X enjoys several properties.

Lemma 4.1. *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:*

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

*(II) $\mathcal{I}_W \equiv \mathcal{I}_X$, 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**).*

*(III) $Freq_X(U) = Freq_W(U)$, for all $U \in \Sigma^k \setminus \{W[i \mathinner{.\,.} i+k-1] : C[i] = 1\}$ (**P2**).*

*(IV) 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**).*

*(V) $0 \le |X| \le \lceil \frac{n-k+1}{2} \rceil \cdot k + \lfloor \frac{n-k+1}{2} \rfloor$ 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
       rightmost 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 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. **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 and 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 Figure 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, $n-k$ be even, and $\mathcal{S} = \{1, 3, 5, \ldots, n-k-1\}$. $C[0] = 0$ 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 4.2).

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

Base case: $j = j_{\min}$. By Lines 3 and 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 $W[0\,..\,j]$.

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 $W[f\,..\,f+k-2] = W[j-k+1\,..\,j-1]$ then $X_j = X_{j-1}W[j]$, and once again X_j is evidently optimal. Otherwise, $X_j = X_{j-1}\#W[j-k+1\,..\,j]$. 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 $W[j-k+1\,..\,j]$ 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 $W[f\,..\,f+k-2] \neq W[j-k+1\,..\,j-1]$). Then X_j is optimal. □

*Example 4.2 (Illustration of **P3**).* Let $k = 4$. We construct the order-3 de Bruijn sequence $W = \mathsf{baaabbbaba}$ of length $n = 10$ over alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$, and choose $\mathcal{S} = \{1, 3, 5\}$. TFS-ALGO constructs:

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

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.

Let us now show the main result of this section.

Theorem 3.1. *Let W be a string of length n over $\Sigma = \{1, \ldots, n^{O(1)}\}$. Given $k < n$ and $\mathcal{S}$,* TFS-ALGO *solves Problem 1 in $O(kn)$ time, which is worst-case optimal. An $O(n)$-sized representation of X can be built in $O(n)$ time.*

Proof. For the first part inspect TFS-ALGO. Lines 2–4 can be realized in $O(n)$ time. The *while* loop in Line 5 is executed no more than n times, and every operation inside the loop takes $O(1)$ time except for Line 13 and Line 17 which take $O(k)$ time. Correctness and optimality follow directly from Lemma 4.1 (**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 4.1 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 [19]. All other operations take in total linear time in n. Thus there exists an $O(n)$-sized representation of X and it is constructible in $O(n)$ time. □

5 PFS-ALGO

Lemma 4.1 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 **Π1** (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.

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 [24]. Specifically our algorithm: (I) Computes the string X using Theorem 3.1. (II) Constructs a collection $\mathcal{B}'$ of strings, each of two letters (two ranks); the first (resp., second) letter is the lexicographic rank of the length-ℓ prefix (resp., suffix) of each string in the collection $\mathcal{B} = \{X_1, \ldots, X_N\}$. (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 5.1 (Illustration of the Workings of PFS-ALGO*).* Let $\ell = k - 1 = 3$ and

$$X = \mathsf{aabaa\#aaacbcbbba\#baabbacaab}.$$

The collection $\mathcal{B}$ is comprised of the following substrings: $X_1 = \mathsf{aabaa}$, $X_2 = \mathsf{aaacbcbbba}$, and $X_3 = \mathsf{baabbacaab}$. The collection $\mathcal{B}'$ is comprised of the following two-letter strings: 23, 14, 32. To construct B', we first find the length-3 prefix and the length-3 suffix of each X_i, $i \in [1, 3]$, which leads to a collection {aab, baa, aaa, bba}. Then, we sort the collection lexicographically to obtain {aaa, aab, baa, bba}, and last we replace each X_i, $i \in [1, 3]$, with the lexicographic ranks of its length-3 prefix and length-3 suffix. For instance, X_1 is replaced by 23. After that, a shortest string containing all elements of $\mathcal{B}'$ as distinct substrings is computed as: $14 \cdot 232$. This shortest string is mapped back to the solution $Y = \mathsf{aaacbcbbba\#aabaabbacaab}$. Note, Y contains one occurrence of # and has length 23, while X contains 2 occurrences of # and has length 27.

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:

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

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 4.1, **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 [24]. However, if all strings are of length two, the SCS problem admits a linear-time solution [24]. We exploit this crucial detail positively to show a linear-time solution to the FO-SSM problem in Lemma 5.3. In order to arrive to this result, we first adapt the SCS linear-time solution of [24] to our needs (see Lemma 5.2) and plug this solution into Lemma 5.3.

Lemma 5.2. *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 [16]. 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 into 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 $\text{IN}(u) = \text{OUT}(u) = 0$ 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. □

Lemma 5.3. *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.*

Proof. 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 Longest Common Prefix (LCP) data structure over the concatenation of strings in $\mathcal{B}$ [19]. 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 5.2. The statement follows. □

Thus, PFS-ALGO applies Lemma 5.3 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 5.4. *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**, **Π1**, and **P2-P4**.*

Proof. **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 there exists an injective function f from the p-chains of $\mathcal{I}_W$ to the p-chains of $\mathcal{I}_Y$ such that $f(\mathcal{J}_W) \equiv \mathcal{J}_W$ for any p-chain $\mathcal{J}_W$ of $\mathcal{I}_W$ (**Π1** is preserved). **P3** also holds trivially for Y as no occurrence of # is added. Since $|Y| \leq |X|$, for **P4**, it suffices to note that the construction of W in the proof of tightness in Lemma 4.1 (see also Example 4.2) 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 5.3 that computes a shortest such string. □

Let us now show the main result of this section.

Theorem 3.2. *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 3.1. 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 [19]; and implement Lemma 5.3 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 5.4. □

6 MCSR PROBLEM, MCSR-ALGO, AND IMPLAUSIBLE PATTERN ELIMINATION

In the following, we introduce the MCSR problem and prove that it is NP-hard (see Section 6.1). Then, we introduce MCSR-ALGO, a heuristic to address this problem (see Section 6.2). Finally, we discuss how to configure MCSR-ALGO in order to eliminate implausible patterns (see Section 6.3).

6.1 The MCSR Problem

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 could be an occurrence of the sensitive pattern. For example, applying this process to the first # in X shown in Figure 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 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 level of distortion bounded by a parameter θ in Z. The cost of an occurrence of a τ-ghost at a certain position is given by function *Ghost*, while function *Sub* assigns a distortion weight to each letter that could replace a #. Both functions will be described in further detail below.

To preserve privacy, we require separator replacements not to reinstate sensitive patterns. This is the MCSR problem, a restricted version of which is presented in Problem 4. The restricted version is referred to as $\text{MCSR}_{k=1}$ and differs from MCSR in that it uses $k = 1$ for the pattern length instead of an arbitrary value $k > 0$. $\text{MCSR}_{k=1}$ is presented next for simplicity and because it is used in the proof of Lemma 6.1. Lemma 6.1 implies Theorem 3.3.

Problem 4 ($\text{MCSR}_{k=1}$). *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:*

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

Lemma 6.1. *The $\text{MCSR}_{k=1}$ problem is NP-hard.*

Proof. We reduce the NP-hard *Multiple Choice Knapsack* (MCK) problem [45] 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 (Equation (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) $\sum_{j\in C_i} x_{ij} = 1, i = 1, \ldots \delta$, and (III) $x_{ij} \in \{0, 1\}, i = 1, \ldots, \delta, j \in C_i$.

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 I_{MCK} to an instance $\text{I}_{\text{MCSR}_{k=1}}$ 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 $S_{\text{I}_{\text{MCK}}}$ to the given instance I_{MCK} and a solution $S_{\text{I}_{\text{MCSR}_{k=1}}}$ to the instance $\text{I}_{\text{MCSR}_{k=1}}$.

We first show that if $S_{\text{I}_{\text{MCK}}}$ is a solution to I_{MCK}, then $S_{\text{I}_{\text{MCSR}_{k=1}}}$ is a solution to $\text{I}_{\text{MCSR}_{k=1}}$. Since the elements in $S_{\text{I}_{\text{MCK}}}$ 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

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

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

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

(i.e., condition I in Problem 4 is satisfied). Since the elements in $S_{\mathrm{I}_{\mathrm{MCK}}}$ 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 $\sum_{i\in[1,\delta]} \sum_{j\in C_i} (1 - \mathrm{Sub}(\mu(i), \alpha_i)) \cdot x_{ij} \geq \delta - \theta$, which implies

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

(i.e., condition II in Problem 4 is satisfied). $S_{\mathrm{I}_{\mathrm{MCSR}_{k=1}}}$ is thus a solution to $\mathrm{I}_{\mathrm{MCSR}_{k=1}}$.

We finally show that, if $S_{\mathrm{I}_{\mathrm{MCSR}_{k=1}}}$ is a solution to $\mathrm{I}_{\mathrm{MCSR}_{k=1}}$, then $S_{\mathrm{I}_{\mathrm{MCK}}}$ is a solution to $\mathrm{I}_{\mathrm{MCK}}$. Since each $\#_i$, $i \in [1, \delta]$, is replaced by a single letter α_i in $S_{\mathrm{I}_{\mathrm{MCSR}_{k=1}}}$, exactly one element will be selected from each class C_i (i.e., conditions II-III of MCK are satisfied). Since the letters in $S_{\mathrm{I}_{\mathrm{MCSR}_{k=1}}}$ satisfy condition I of Problem 4, 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 $S_{\mathrm{I}_{\mathrm{MCSR}_{k=1}}}$ satisfies $\sum_{i:Y[i]=\#} \mathrm{Sub}(i, Z[i]) = \sum_{i\in[1,\delta]} \sum_{j\in C_i} \mathrm{Sub}(\mu(i), \alpha_{ij}) \cdot x_{ij} \leq \theta$, 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, $S_{\mathrm{I}_{\mathrm{MCK}}}$ is a solution to $\mathrm{I}_{\mathrm{MCK}}$. The statement follows. □

Lemma 6.1 implies the main result of this section.

Theorem 3.3. *The* MCSR *problem is NP-hard.*

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 [18].

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 #. As will be explained in Section 6.3, weights larger than θ are also assigned to letters which would lead to implausible substrings [27] if they replaced #s.

6.2 MCSR-ALGO

We next present MCSR-ALGO, a non-trivial heuristic that exploits the connection of the MCSR and MCK [40] problems. We start with a high-level description of MCSR-ALGO:

(I) Construct 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) Create 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) Solve the instance of MCK and translate 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 6.2 below states the running time of an efficient implementation of MCSR-ALGO.

Lemma 6.2. MCSR-ALGO *runs in* $O(|Y| + k\delta\sigma + \mathcal{T}(\delta, \sigma))$ *time, where* $\mathcal{T}(\delta, \sigma)$ *is the running time of the MCK algorithm for* δ *classes with* $\sigma + 1$ *elements each.*

Proof. 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 C of *candidate τ-ghosts* as follows. The candidates are at most $(|Y| - k + 1 - k\delta) + (k\delta\sigma) = O(|Y| + k\sigma\delta)$ 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 $O(|Y| + k\delta\sigma)$ 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 C. Let S_{ij} denote 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 C. $\text{Freq}_Y(U)$, for all U occurring in Y, can be computed in $O(|Y|)$ total time using the suffix tree of Y [19].

(II) If $\max \text{Freq}_Z(U) < \tau$, U by definition can never become τ-ghost in Z, and we thus exclude it from 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 $O(|Y| + k\delta\sigma)$ time [19]. We then spell $S_{i1}, \ldots, S_{i\sigma}$, for all i, in the generalized suffix tree in $O(k\sigma)$ time per i. We exploit suffix links to spell the length-k substrings of S_{ij} in $O(k)$ time and memorize the maximum number of occurrences of U caused by replacing the ith occurrence of # among all j. We represent set C on the generalized suffix tree by marking the corresponding nodes, and we denote this representation by $T(C)$. 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 $T(C)$. 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 $T(C)$. 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 $T(C)$ 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 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 $\mathcal{O}(|Y| + k\delta\sigma)$.

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 $\mathcal{O}(|Y| + k\delta\sigma + \mathcal{T}(\delta, \sigma))$. □

6.3 Eliminating Implausible Patterns

We present the notion of implausible substring and explain how we can ensure that implausible patterns do not occur in Z, as a result of applying the MCSR-ALGO algorithm to string Y.

Consider, for instance, an input string $Y = \ldots$ a#c $\ldots$ that models the movement of an individual, and the string abc, which is created as a substring of Z when we replace # with b. Consider further that an individual can, generally, not go from a to c through b, or that it is highly unlikely for them to do so. We call a substring such as abc *implausible*. Clearly, if abc occurs in Z, it may be possible for an attacker to infer that b replaced #, and then infer a sensitive pattern by "undoing" **R1** as explained in Section 6.1. In order to effectively model this scenario, we define implausible patterns based on a statistical significance measure for strings [7, 15, 42]. The measure is defined as follows [15]:

$$z_W(U) = \frac{\text{Freq}_W(U) - \mathbb{E}_W[U]}{\max(\sqrt{\mathbb{E}_W[U]}, 1)},$$

where U is a string with $|U| > 2$, W is the reference string, and

$$\mathbb{E}_W[U] = \begin{cases} \frac{\text{Freq}_W(U[0\,..\,|U|-2]) \cdot \text{Freq}_W(U[1\,..\,|U|-1])}{\text{Freq}_W(U[1\,..\,|U|-2])}, & \text{Freq}_W(U[1\,..\,|U|-2]) > 0 \\ 0, \text{otherwise} \end{cases}$$

is the expected frequency of U in W, computed based on an independence assumption between the event "$U[0\,..\,|U|-1]$ occurs in W" and "$U[1\,..\,|U|-1]$ occurs in W." The measure z_W is a normalized version of the standard score of U, based on the fact that the variance $\text{Var}_W[U] \approx \sqrt{\mathbb{E}_W[U]}$ [42]. A small $z_W(U)$ indicates that U occurs less likely than expected, and hence it can naturally be considered as an artefact of sanitization.

Given a user-defined threshold $\rho < 0$, we define a string U as *ρ-implausible* if $z_W(U) < \rho$. The set of ρ-implausible substrings of W can be computed in the optimal $\mathcal{O}(|\Sigma| \cdot |W|)$ time [7]. We use W as the reference string, assuming that it is a good representation of the domain; e.g., a trip (substring) that is ρ-implausible in W is also implausible in general. Alternatively, one could use any other string as reference, impose length constraints on implausible patterns [32, 47], or even directly specify substrings that should not occur in Z based on domain knowledge.

Given the set $\mathcal{U}$ of (ρ-)implausible patterns, we ensure that no # replacement creates $U = U_1 \alpha U_2 \in \mathcal{U}$ in Z, where α is the letter that replaces #, by assigning a weight $\text{Sub}(i, Z[i]) > \theta$, for each $Z[i]$ such that $Y[i] = \#$ and $U_1 \cdot Z[i] \cdot U_2 \in \mathcal{U}$. This guarantees that no replacement leading to an artefact occurrence of an element of $\mathcal{U}$ is performed by MCSR-ALGO. Note, however, that a ρ-implausible pattern may occur in Z as a substring, either because it occurred in a part of W that was copied to Z (e.g., a non-sensitive pattern), or due to the change of frequency of some substrings that are created in Z after the replacement of a #. However, since such ρ-implausible patterns did not contain a # in the first place, they cannot be exploited by an attacker seeking to reverse the construction of Z.

7 ETFS-ALGO

Let U and V be two non-sensitive length-k substrings of W such that U is the t-predecessor of V. Since U and V must occur in the same order in the solution string X_{ED}, the main choice we have to make in order to solve the ETFS problem is whether to:

(I) "merge" U and V when the length-$(k-1)$ suffix of U and the length-$(k-1)$ prefix of V match; or

(II) "interleave" U and V with a carefully selected string over $\Sigma \cup \{\#\}$.

Among operations I and II, for every such pair U and V, we must select the operation that *globally* results in the smallest number of edit operations. Operations I and II can naturally be expressed by means of a regular expression E. In particular, this implies that any instance of the ETFS problem can be reduced to an instance of approximate regular expression matching and thus an algorithm for approximate regular expression matching between E and W [37] can be employed. More formally, given a string W and a regular expression E, the *approximate regular expression matching* problem is to find a string T that matches E with minimal $d_E(W, T)$. The following result is known.

Theorem 7.1 ([37]). *Given a string W and a regular expression E, the approximate regular expression matching problem can be solved in $\mathcal{O}(|W| \cdot |E|)$ time.*

In the following, we define a specific type of a regular expression E. Let us first define the following regular expression:

$$\Sigma^{<k} = (\underbrace{(a_1|a_2|\ldots|a_{|\Sigma|}|\varepsilon)\ldots(a_1|a_2|\ldots|a_{|\Sigma|}|\varepsilon)}_{k-1 \text{ times}}),$$

where $\Sigma = \{a_1, a_2, \ldots, a_{|\Sigma|}\}$ is the alphabet of W and $k > 1$. We also define the following regular-expression gadgets, for a letter $\# \notin \Sigma$:

$$\oplus = \#(\Sigma^{<k}\#)^*, \quad \ominus = (\Sigma^{<k}\#)^*, \quad \otimes = (\#\Sigma^{<k})^*.$$

Intuitively, the gadget $\oplus$ represents a string we may choose to include in the output in an effort to minimize the edit distance between W and the solution string X_{ED}. It should be clear that the length of $\oplus$ is in $\mathcal{O}(k|\Sigma|)$ and that $\oplus$ cannot generate any length-k substring over Σ. Furthermore, inserting $\oplus$ in E cannot create any sensitive or non-sensitive pattern due to the occurrences of # on both ends of $\oplus$. The gadgets $\ominus$ and $\otimes$ are similar to $\oplus$. They are added in the beginning and at the end of E, respectively. This is because E should not start or end with # as this would only increase the edit distance to W. As it will be explained later, to construct E, we also make use of the | operator. Intuitively, the | operator represents the choice we make between operation "merge" or "interleave."

We are now in a position to describe ETFS-ALGO, an algorithm for solving the ETFS problem. ETFS-ALGO starts by constructing E. Let $(N_1, N_2 \ldots, N_{|\mathcal{I}|})$ be the sequence of non-sensitive length-k substrings as they occur in W from left to right. We first set $E = \ominus N_1$ and then process the pairs of non-sensitive length-k substrings N_i and N_{i+1}, for all $i \in \{1, |\mathcal{I}| - 1\}$. At the ith step, we examine whether or not N_i and N_{i+1} can be merged. If they can, we append to E a regular expression $(A| \oplus N_{i+1})$, where A is obtained by chopping-off the length-$(k-1)$ prefix of N_{i+1} (that is, the remainder of N_{i+1} after merging it with N_i). Otherwise, we append $\oplus N_{i+1}$ to E. Intuitively, using A corresponds to choosing "merge" and $\oplus N_{i+1}$ to choosing "interleave." After examining each pair N_i and N_{i+1}, we append $\otimes$ to E. This concludes the construction of E. Note how, for any combination of choices, N_{i+1} will always appear in the string obtained.

Next, ETFS-ALGO employs Theorem 7.1 to construct X_{ED}. In particular, it finds a string T that matches E with minimal $d_E(W, T)$. Last, it sets $X_{\text{ED}} = T$. We arrive at the main result of this section.

Theorem 3.4. *Let W be a string of length n over an alphabet Σ. Given $k < n$ and $\mathcal{S}$,* ETFS-ALGO *solves Problem 3 in $\mathcal{O}(k|\Sigma|n^2)$ time.*

Proof. Constructing E can be done in $\mathcal{O}(n + kn + |E|) = \mathcal{O}(k|\Sigma|n)$ time, since: (I) The non-sensitive length-k substrings of W can be obtained in $\mathcal{O}(n)$ time, by reading W from left to right and checking $\mathcal{S}$. (II) Checking whether N_i and N_{i+1} are mergeable takes $\mathcal{O}(k)$ time via letter comparisons, and it is performed in each of the $\mathcal{O}(n)$ steps. (III) The length is $|E| = \mathcal{O}(kn + k|\Sigma|n) = \mathcal{O}(k|\Sigma|n)$. This is because E contains at most n occurrences of non-sensitive length-k substrings, at most n occurrences of $\oplus$, and one occurrence of each of $\ominus$ and $\otimes$ and because the lengths of $\oplus$, $\ominus$ and $\otimes$ are $\mathcal{O}(k|\Sigma|)$.

Computing T from W and E can be performed in $\mathcal{O}(|W| \cdot |E|) = \mathcal{O}(n \cdot |E|)$ time using Theorem 7.1. Thus ETFS-ALGO takes $\mathcal{O}(k|\Sigma|n^2)$ time in total.

The correctness of ETFS-ALGO follows from the fact that by construction: (I) T does not contain any sensitive pattern, so **C1** is satisfied; (II) T satisfies **P1** and **P2** as no length-k substring over Σ (other than the non-sensitive ones) is inserted in E; (III) All strings satisfying **C1**, **P1**, and **P2** can be obtained by E, since they must have the same t-chain of non-sensitive patterns over Σ^* as W, interleaved by length-k substrings that are on $(\Sigma \cup \#)^*$ but *not* on Σ^*; and (IV) the minimality on edit distance is guaranteed by Theorem 7.1. The statement follows. □

A factor of $|\Sigma|$ can be shaved from $\mathcal{O}(k|\Sigma|n^2)$ via dynamic programming [9], albeit it seems unlikely to yield a strongly subquadratic time bound [8]. In any case, as our experiments show, TFS-ALGO, which runs in $\mathcal{O}(kn)$ time, outputs optimal or near-optimal solutions in practice.

Example 7.2 (Illustration of the Workings of ETFS-ALGO*).* Let $W = \mathsf{aaabbaabaccbbb}$, $k = 4$, and the set of sensitive patterns be $\{\mathsf{aabb}, \mathsf{abba}, \mathsf{bbaa}, \mathsf{baab}, \mathsf{ccbb}\}$. The sequence of non-sensitive patterns is thus $(N_1, \ldots, N_6) = (\mathsf{aaab}, \mathsf{aaba}, \mathsf{abac}, \mathsf{bacc}, \mathsf{accb}, \mathsf{cbbb})$. Given that $k = 4$ and $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, ETFS-ALGO constructs the following gadgets,

$$\oplus = \#(\Sigma^{<4}\#)^* = \#(((\mathsf{a}|\mathsf{b}|\mathsf{c}|\varepsilon)(\mathsf{a}|\mathsf{b}|\mathsf{c}|\varepsilon)(\mathsf{a}|\mathsf{b}|\mathsf{c}|\varepsilon))\#)^*$$

$$\ominus = (\Sigma^{<4}\#)^* = (((\mathsf{a}|\mathsf{b}|\mathsf{c}|\varepsilon)(\mathsf{a}|\mathsf{b}|\mathsf{c}|\varepsilon)(\mathsf{a}|\mathsf{b}|\mathsf{c}|\varepsilon))\#)^*$$

$$\otimes = (\#\Sigma^{<4})^* = (\#((\mathsf{a}|\mathsf{b}|\mathsf{c}|\varepsilon)(\mathsf{a}|\mathsf{b}|\mathsf{c}|\varepsilon)(\mathsf{a}|\mathsf{b}|\mathsf{c}|\varepsilon)))^*$$

and sets $E = \ominus N_1 = \ominus\mathsf{aaab}$. Then, it iterates over each pair of consecutive non-sensitive length-k substrings in the order they appear in W (i.e., pair (N_i, N_{i+1}) is considered in Step $i \in [1, 5]$) and the regular expression E is updated, as detailed below.

In Step 1, ETFS-ALGO considers the pair $(N_1, N_2) = (\text{aaab}, \text{aaba})$. Observe that in this case N_1 and N_2 can be merged, since the length-3 suffix of N_1 and the length-3 prefix of N_2 match. Thus, $(A|N_2) = (\text{a}| \oplus \text{aaba})$ is appended to E. Recall that when merging, we chop off the length-$(k-1)$ prefix of $N_{i+1} = N_2$ (because we have merged it already) and write down what is left of N_2 (a in this case) before |. Thus, $E = \ominus\text{aaab}(\text{a}| \oplus \text{aaba})$.

In Step 2, ETFS-ALGO considers $(N_2, N_3) = (\text{aaba}, \text{abac})$. Again, N_2 and N_3 can be merged. Thus, $(\text{c}| \oplus \text{abac})$ is appended into E, which leads to $E = \ominus\text{aaab}(\text{a}| \oplus \text{aaba})(\text{c}| \oplus \text{abac})$.

In Steps 3 and 4, ETFS-ALGO considers the pairs $(N_3, N_4) = (\text{abac}, \text{bacc})$ and $(N_4, N_5) = (\text{bacc}, \text{accb})$, respectively. Since the patterns in each pair can be merged, the algorithm appends into E the regular expression $(\text{c}| \oplus \text{bacc})$ and $(\text{b}| \oplus \text{accb})$, for the first and second pair, respectively. This leads to $E = \ominus\text{aaab}(\text{a}| \oplus \text{aaba})(\text{c}| \oplus \text{abac})(\text{c}| \oplus \text{bacc})(\text{b}| \oplus \text{accb})$.

In Step 5, ETFS-ALGO considers the last pair $(N_5, N_6) = (\text{accb}, \text{cbbb})$, which cannot be merged, and appends $\oplus$cccb to E. Since there is no other pair to be considered, $\otimes$ is also appended to E, leading to:

$$E = \ominus\underline{\text{aaab}}(\text{a}|\underline{\oplus\text{aaba}})(\underline{\text{c}}| \oplus \text{abac})(\underline{\text{c}}| \oplus \text{bacc})(\underline{\text{b}}| \oplus \text{accb})\underline{\oplus\text{cbbb}} \otimes .$$

At this point, ETFS-ALGO employs Theorem 7.1 to find the following string T that matches E (the choices that were made in the construction of T are underlined in E and $\ominus$, $\oplus$, $\otimes$ are matched by the empty string):

$$T = \text{aaab\#aabaccb\#cbbb},$$

with minimal $d_E(T, W) = 4$. Last, ETFS-ALGO returns $X_{\text{ED}} = T$.

Note that $X_{\text{ED}} = T$ in Example 7.2 does not contain any sensitive pattern and that all non-sensitive patterns of W appear in T in the same order and with the same frequency as they appear in W. Note also that, for the same instance, TFS-ALGO would return string $X =$aaabaccb#cbbb with $d_E(W, X) = 5 > d_E(W, X_{\text{ED}}) = 4$ and $|X| = 13 < |X_{\text{ED}}| = 17$.

8 EXPERIMENTAL EVALUATION

We evaluate our algorithms in terms of *effectiveness* and *efficiency*. Effectiveness is measured based on data utility and number of implausible patterns. Efficiency is measured based on runtime.

Evaluated Algorithms. First, we consider the pipeline TFS-ALGO→ PFS-ALGO→MCSR-ALGO, referred to as TPM. Given a string W over Σ, TPM sanitizes W by applying TFS-ALGO, PFS-ALGO, and then MCSR-ALGO. MCSR-ALGO uses the $O(\delta\sigma\theta)$-time algorithm of [40] for solving the MCK instances. The final output is a string Z over Σ. MCSR-ALGO is configured with an empty set $\mathcal{U}$ (i.e., it may lead to implausible patterns that are created in Z after the replacement of a #).

Among the related works discussed in Section 2.1, we compared TPM against the PH heuristic [27]. This is because we found PH to be the closest to our setting, and, moreover, because it outperforms other related sequence sanitization methods [1, 25] (see Section 2.1 for details). We also compared TPM against a greedy baseline referred to as BA, in terms of data utility and efficiency. 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, BA 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. Note that this letter replacement should not introduce any other sensitive pattern in Z_{BA}. 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. Note that BA does not preserve the frequency of

Table 3. Characteristics of Datasets and Values Used (Default Values are in Bold)

Dataset	**Data domain**	**Length** n	**Alphabet size** $\|\Sigma\|$	**# sensitive patterns**	**# sensitive positions** $\|\mathcal{S}\|$	**Pattern length** k	**Implausible pat. threshold** ρ
OLD	Movement	85,563	100	[30, 240] (**60**)	[600, 6103]	[3, 7] (**4**)	[−2, −0.1] (**−1**)
TRU	Transportation	5,763	100	[30, 120] (**10**)	[324, 2410]	[2, 5] (**4**)	[−3, −0.1] (**−4**)
MSN	Web	4,698,764	17	[30, 120] (**60**)	[6030, 320480]	[3, 8] (**4**)	[−6, −3] (**−1**)
DNA	Genomic	4,641,652	4	[25, 500] (**100**)	[163, 3488]	[5, 15] (**13**)	[−4.5, −2.5] (**−2.5**)
SYN	Synthetic	20,000,000	10	[10, 1000] (**1000**)	[10724, 20171]	[3, 6] (**6**)	-
SYN_{BIN}	Synthetic	1,000	2	[4, 32] (**16**)	[16, 128]	[4, 7] (**4**)	-

non-sensitive patterns, and thus, unlike TPM, it can incur τ-lost patterns. We also implemented a similar baseline that replaces the letter in R that has the smallest frequency in Z_{BA} with another letter that is not contained in R and has the largest frequency in Z_{BA}, but omit its results as it was worse than BA.

In addition, we consider the pipelines TFS-ALGO→MCSR-ALGO and TFS-ALGO→MCSRI-ALGO, referred to as TM and TMI, respectively. With MCSRI-ALGO we refer to the configuration of MCSR in which there is a non-empty set $\mathcal{U}$ of ρ-implausible patterns that must not occur in the output string Z. We omit PFS-ALGO from the TM and TMI pipelines to avoid the elimination of some implausible patterns due to re-ordering of blocks of non-sensitive patterns that is performed by PFS-ALGO.

Last, we consider ETFS-ALGO, which we compare to TFS-ALGO, to demonstrate that the latter is a very effective heuristic for the ETFS problem.

Experimental Data. We considered the following publicly available datasets used in [1, 11, 25, 27, 31]: 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 3 for the characteristics of these datasets and the parameter values used in experiments, unless stated otherwise.

Experimental Setup. The sensitive patterns were selected randomly among the frequent length-k substrings at minimum support τ following [25, 27, 31]. We used the fairly low values ($\tau = 10$ for TRU, SYN, and SYN_{BIN}; $\tau = 20$ for OLD and DNA; and $\tau = 200$ for MSN), to have a wider selection of sensitive patterns. In MCSR-ALGO, we 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. In MCSRI-ALGO, we instead set a weight ∞ for each letter replacement that does not create a sensitive pattern or an implausible pattern of length k.

In PH, we used a minimum frequency threshold of $\tau = 1$ to ensure that sensitive patterns will not occur as subsequences (and hence nor as substrings) in the output. We also transformed the input string into a collection of strings and provided the collection as input to PH. This is because, although in principle PH can be applied to a single string, as in Example 2.1, this was not possible for any of the datasets of Table 3. In fact, as it will be shown later, PH did not terminate within 12 hours, even for very short strings of length 25 that took milliseconds to be sanitized by our algorithms. The reason is that PH requires finding all occurrences of every sensitive pattern in the string and computing changes to the set of non-sensitive frequent sequential patterns incurred by permutation and deletion. When $\tau = 1$ and for reasonably long strings, this is a very computationally intensive task. This observation agrees with the findings in [27] and similar findings were reported for other sanitization algorithms [1, 25].

Therefore, to be able to compare with PH, we converted a long string to a collection of short strings (i.e., the type of dataset that PH was designed for). Specifically, we created a collection of strings $W_1, W_2, \ldots, W_m$ from a string W, such that $W = W_1 \cdot W_2 \cdot \ldots \cdot W_m$ and $|W_i| = r$, for $i \in [1, m]$, and then we applied PH to the collection. In our experiments, we varied r in $[5, 25]$ and used $r = 15$ as the default value. The smallest value $r = 5$ was selected to enable the hiding of sensitive patterns of length $k = 5$ that we used; the largest value $r = 25$ was selected empirically. PH took much longer as we increased r and did not terminate within 12 hours for $r = 25$. After applying PH, we obtained a sanitized collection of strings $W_1', W_2', \ldots, W_m'$ and constructed a final string $I = W_1' \cdot W_2' \cdot \ldots \cdot W_m'$ by concatenating the strings in the sanitized collection. Note that we favor PH by neglecting the possibility that sensitive patterns may be created when concatenating the strings in the sanitized collection.

To capture the utility of sanitized data, we used the *(frequency) distortion* measure

$$\sum_U (\mathrm{Freq}_W(U) - \mathrm{Freq}_Z(U))^2,$$

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 [41]).

We also measured the number of τ-ghost and τ-lost patterns in Z following [25, 27, 31], where a pattern U is $\tau-$*lost* in Z if and only if $\mathrm{Freq}_W(U) \geq \tau$ but $\mathrm{Freq}_Z(U) < \tau$. 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 [25, 27, 31]. Unlike BA, by design TPM *does not* incur any τ-lost pattern, as TFS-ALGO and PFS-ALGO preserve frequencies of non-sensitive patterns, and MCSR-ALGO can only increase pattern frequencies.

To examine the benefit of using MCSRI-ALGO instead of MCSR-ALGO when implausible patterns need to be eliminated, we measured the percentage of ρ-implausible patterns of length k that may occur in Z, when a letter replaces a #. Clearly, the percentage is 0 when MCSRI-ALGO is used, and a large percentage for MCSR-ALGO implies that it is beneficial to use MCSRI-ALGO instead.

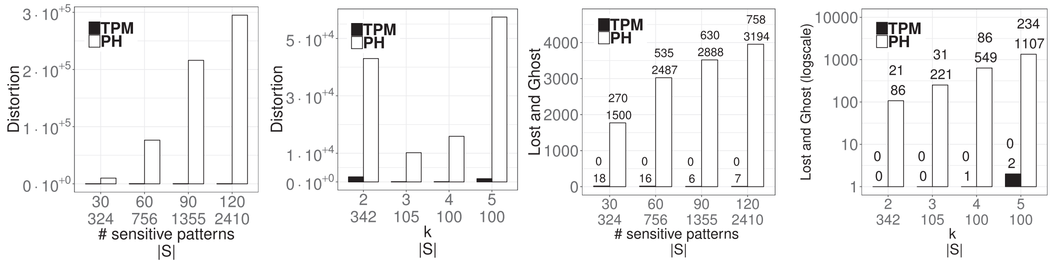
To capture the effectiveness of TFS-ALGO in terms of constructing a string X that is at small edit distance from W (see the ETFS problem), we used the *Edit Distance Relative Error*, defined as

$$\frac{d_E(W, X) - d_E(W, X_{\mathrm{ED}})}{d_E(W, X_{\mathrm{ED}})}.$$

All experiments ran on a Desktop PC with an Intel Xeon E5-2640 at 2.66GHz and 16GB RAM. Our source code is written in C++ and is accessible from https://bitbucket.org/stringsanitization/stringsanitizationtkdd/. The code for PH is also written in C++ and was provided by the authors of [27]. The results presented below have been averaged over 10 runs.

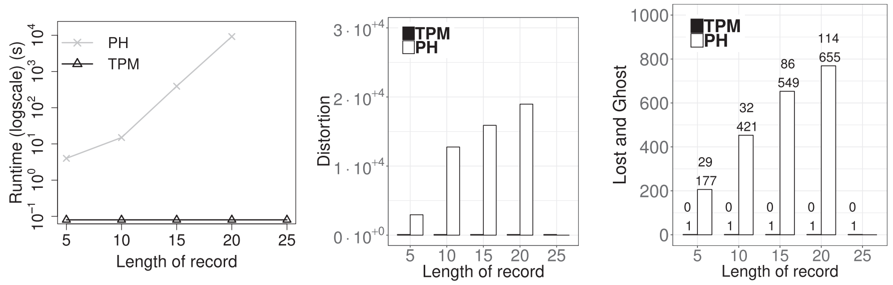
8.1 TPM vs. PH

Data Utility. We first demonstrate that TPM substantially outperformed PH in terms of distortion. This suggests that TPM is a much better method for preserving utility in tasks based on the frequency of substrings (e.g., [41]). Figure 2(a) shows that, for varying number of sensitive patterns, TPM incurred on average 477 (and up to 1,045) times lower distortion than PH did. These results are expected because PH applies permutation and/or deletion to eliminate all occurrences



(a) TRU **(b)** TRU **(c)** TRU **(d)** TRU

Fig. 2. Distortion vs. (a) number of sensitive patterns and their total number $|\mathcal{S}|$ of occurrences in W (first two lines on the X axis), and (b) length of sensitive patterns k (and $|\mathcal{S}|$). Total number of τ-lost and τ-ghost patterns vs. (c) length of sensitive patterns k, and (d) length of sensitive patterns k (and $|\mathcal{S}|$). $\frac{x}{y}$ on the top of each bar denotes x τ-lost and y τ-ghost patterns.



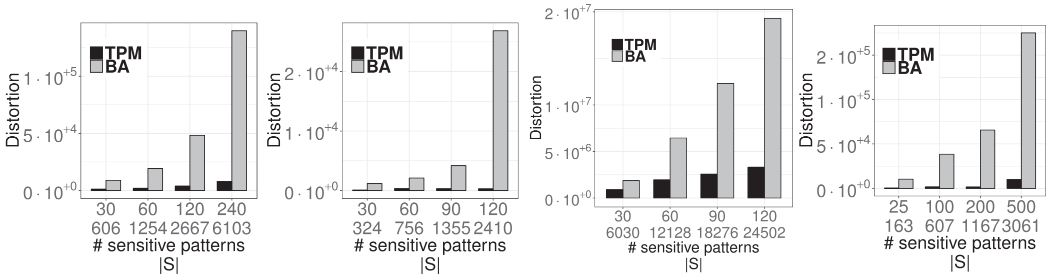
(a) TRU **(b)** TRU **(c)** TRU

Fig. 3. (a) Runtime, (b) distortion, and (c) total number of τ-lost and τ-ghost patterns vs. length of the records of the input dataset to PH. Note that PH did not terminate within 12 hours when $r = 25$.

of a sensitive pattern as a subsequence from the sanitized output, whereas only the occurrences in which the pattern is comprised of consecutive letters (i.e., the sensitive pattern occurs as a substring) should be eliminated. This "overprotection" incurs distortion unnecessarily and severely harms utility, particularly when there are more sensitive patterns. Indeed, Figure 2(a) shows that PH becomes less effective as the number of sensitive patterns increases. In addition, TPM incurred substantially less distortion than PH for all tested values of k. Figure 2(b) shows that TPM incurred on average 78 (and up to 169) times lower distortion than PH. This is again because our setting calls for hiding occurrences of sensitive patterns as substrings and, in this setting, PH overprotects data unnecessarily.

We now demonstrate that TPM allows substantially more accurate frequent substring mining than PH. Figure 2(c) shows that, for varying number of sensitive patterns, the number of τ-lost and τ-ghost patterns for TPM was on average 376 (and up to 586 times) lower compared to that of PH. Quantitatively similar results were obtained for varying k, as can be seen in Figure 2(d). Specifically, the number of τ-lost and τ-ghost patterns for TPM was at least 21 (and up to 234) times lower than that of PH. Note that TPM creates no τ-lost patterns by design and it created no more than 2 τ-ghost patterns in the experiments of Figure 2(d), while PH created up to 234 τ-lost and 1107 τ-ghost patterns.

Impact of r on Efficiency. We demonstrate the runtime of PH as a function of r, the length of records in the collection of records $W_1, W_2, \ldots, W_m$ that was created from a string dataset W and given as input to PH. As can be seen in Figure 3(a), the runtime of PH increased from 4 seconds



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

Fig. 4. Distortion vs. number of sensitive patterns and their total number $|\mathcal{S}|$ of occurrences in W (first two lines on the X axis).

when $r = 5$ to 2.5 hours when $r = 20$. Also, PH did not terminate within 12 hours for $r = 25$. This shows why it was not feasible to apply PH directly to an entire string dataset of Table 3, and we needed to construct a collection of sequences instead. As mentioned in "Experimental setup" above, the reason is that PH needs much time to hide all occurrences of sensitive patterns as subsequences for large strings, particularly when $\tau = 1$, which is needed to reduce the frequency of sensitive patterns (substrings) to zero. On the other hand, TPM required less than a second to process W. Note that the results reported for TPM are the same for all values of r, because r is not an input parameter to TPM.

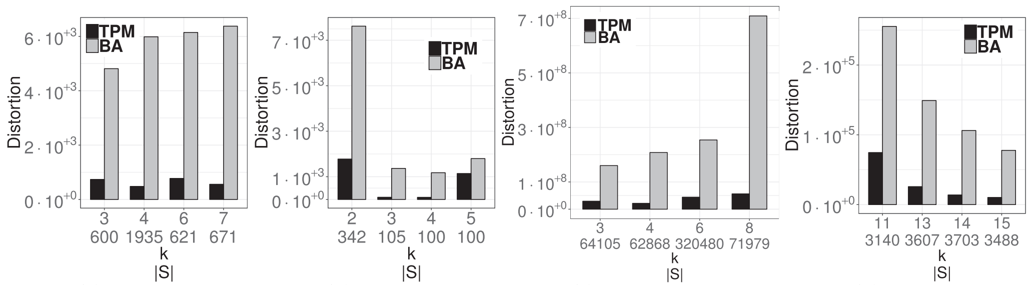
Impact of r on Data Utility. We demonstrate that TPM substantially outperforms PH, for all tested values of r, both in terms of distortion and number of τ-lost and τ-ghost patterns. Specifically, TPM incurred on average 169 (and up to 201) times lower distortion than PH. Also, it created only 1 τ-lost pattern, while PH created at least 29 τ-lost and 421 τ-ghost patterns. The reason that PH gets worse when r increases is because a longer record implies that there are generally more occurrences of sensitive patterns (as subsequences) that PH needs to hide, and this requires more substantial changes to the input data. Note that the results reported for TPM are the same for all values of r, because r is not an input parameter to TPM.

8.2 TPM vs. BA

Data Utility. We first demonstrate that TPM incurs *very low distortion*. Figure 4 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, Figure 4 shows that TPM remains effective even in challenging settings, with many sensitive patterns (e.g., the last point in Figure 4(b) where about 42% of the positions in W are sensitive). Figure 5 shows that, for varying k, TPM caused on average 7.6 (and up to 14) times lower distortion than BA over all experiments.

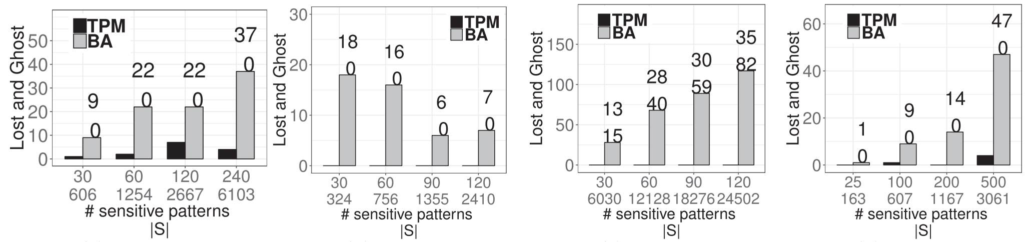
Next, we demonstrate that TPM permits *accurate frequent pattern mining*: Figure 6 shows that TPM led to no τ-lost or τ-ghost patterns for the TRU and MSN datasets. This implies no utility loss for mining frequent length-k substrings with threshold τ. In all other cases, the number of τ-ghosts was on 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). Figure 7 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).

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



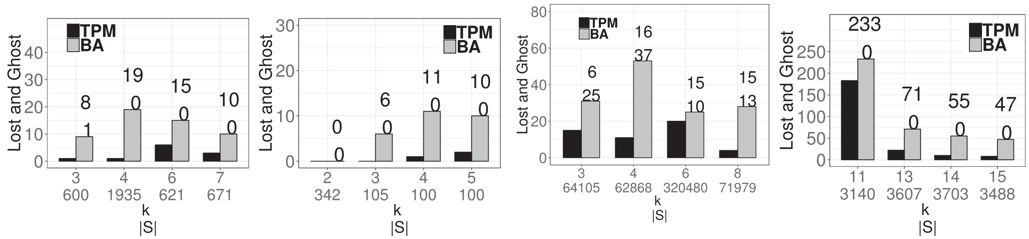
(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

Fig. 5. Distortion vs. length of sensitive patterns k (and $|\mathcal{S}|$).



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

Fig. 6. 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.

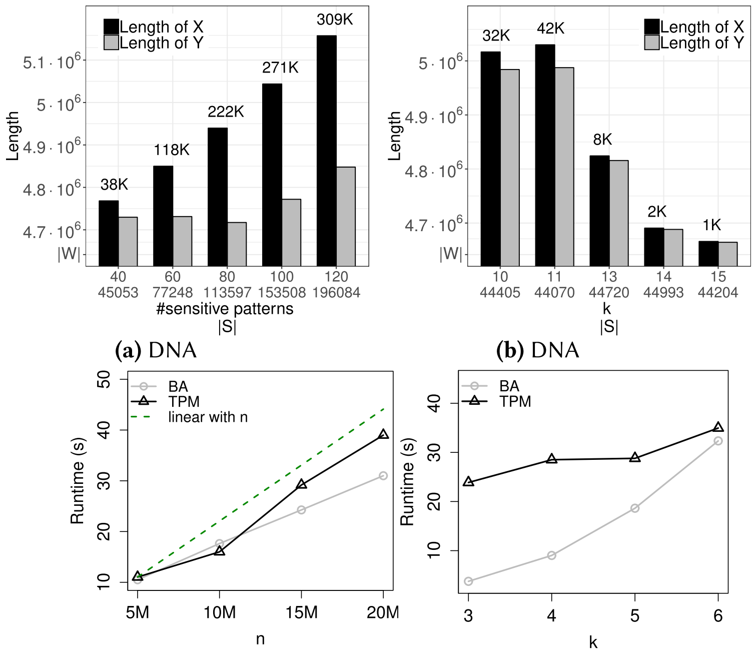


(a) OLD **(b)** TRU **(c)** MSN **(d)** DNA

Fig. 7. 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.

efficient analysis. Figure 8(a) 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 5). 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 Figure 8(b) (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.

Efficiency. We finally measured the runtime of TPM using prefixes of the synthetic string SYN whose length n is 20 million letters. Figure 8(c) (resp., Figure 8(d)) shows that TPM scaled linearly with n (resp., k), as predicted by our analysis in Section 6 (TPM takes $\mathcal{O}(n + |Y| + k\delta\sigma + \delta\sigma\theta) =$



(a) DNA **(b)** DNA

(c) Substr. of SYN **(d)** SYN

Fig. 8. Length of X and Y (output of TFS-ALGO and PFS-ALGO, respectively) for varying: (a) number of sensitive patterns (and $|\mathcal{S}|$), and (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 $|Y| = |Z|$.

$O(kn + k\delta\sigma + \delta\sigma\theta)$ time, since the algorithm of [40] 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.

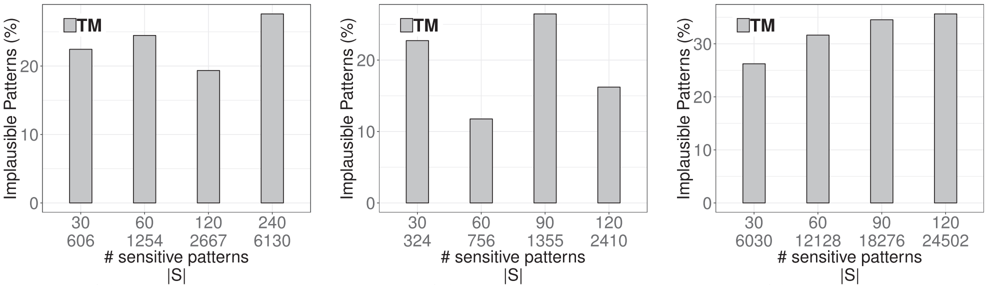
8.3 TM vs. TMI

We compare TM with TMI based on data utility and the number of implausible patterns incurred. The objective of these experiments is to show that TMI is able to produce a string Z that does not contain implausible patterns, while being comparable to TM in terms of the amount of distortion and number of ghost patterns incurred.

We do not report the results of comparing TM with TMI in terms of efficiency, because the runtime of TMI was almost identical to that of TM.

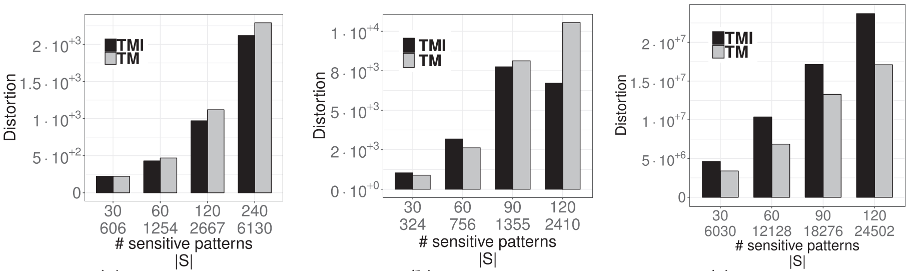
Impact of $|\mathcal{S}|$. We first demonstrate that many implausible patterns may occur as a result of replacing #s with letters, when MCSR is used. This can be seen from Figure 9(a)–9(c), which shows the percentage of implausible patterns incurred by TM, for varying $|\mathcal{S}|$ in OLD, TRU, and MSN, respectively. The percentage is on average 33.08% (and up to 35.63%). The percentage for DNA is 0% (omitted), because this dataset has a very small alphabet size. Thus, in this experiment, MCSR-ALGO and MCSRI-ALGO are essentially the same algorithm. Since TMI is guaranteed to eliminate implausible patterns, its corresponding percentages are zero (omitted).

We then demonstrate that TMI eliminates implausible patterns without incurring substantial utility loss compared to TM. Figures 10 and 11 show that TMI incurred a comparable amount of distortion to TM. Specifically, TMI incurred 8% and 1% less distortion in the case of OLD and TRU datasets and 37% more distortion in the case of MSN. TMI also incurred a similar number of ghosts



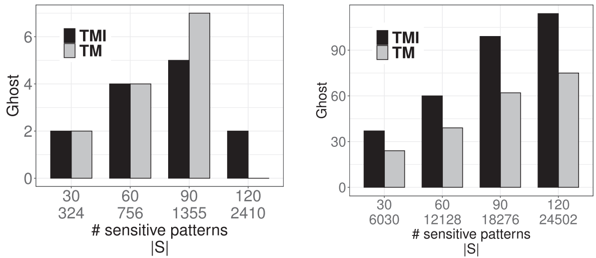
(a) OLD **(b)** TRU **(c)** MSN

Fig. 9. Percentage of implausible patterns vs. number of sensitive patterns (and $|\mathcal{S}|$). The percentages of implausible patterns for DNA are all 0%.



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

Fig. 10. Distortion vs. number of sensitive patterns and their total number $|\mathcal{S}|$ of occurrences in W (first two lines on the X axis).

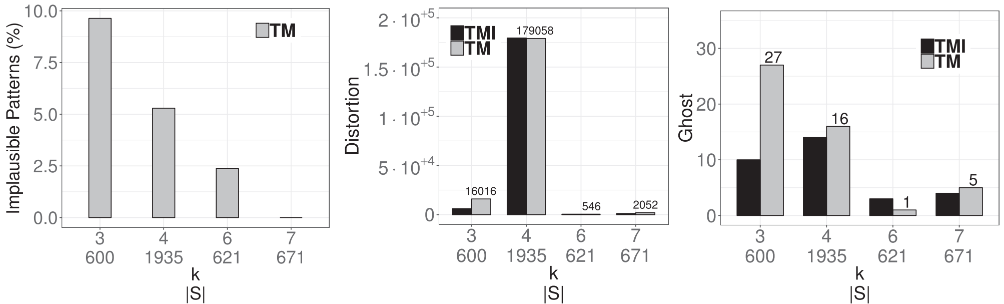


(a) TRU **(b)** MSN

Fig. 11. Number of τ-ghost patterns (the number of τ-lost patterns is zero by design) vs. number of sensitive patterns (and $|\mathcal{S}|$). The number of τ-ghost patterns for OLD is 0.

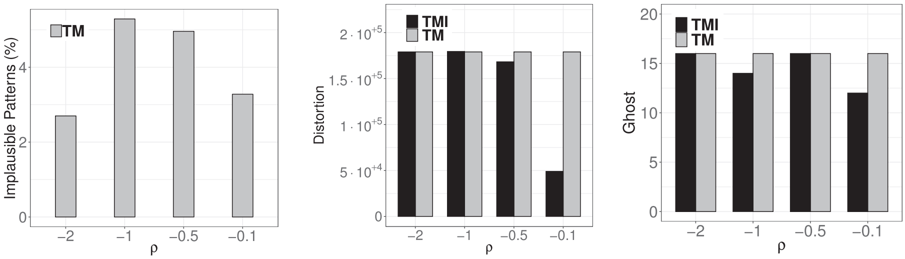
than TM. Specifically, TMI incurred 7.1% fewer ghosts in the case of TRU and 54% more ghosts in the case of MSN. Note that no τ-ghost patterns were incurred in the case of OLD (for both TM and TMI). The worse performance of TMI in the case of the MSN dataset is attributed to its relatively small alphabet size, which makes it more difficult to select a letter replacement that does not incur implausible patterns.

Impact of k. Figure 12(a) shows that the percentage of implausible patterns incurred by TM for the OLD dataset was on average 4.3% (and up to 9.6%). Again, this confirms the need to eliminate implausible patterns in practice. The results for TRU, MSN, and DNA are qualitatively similar and omitted from all remaining experiments.



(a) OLD **(b)** OLD **(c)** OLD

Fig. 12. (a) Percentage of implausible patterns vs. k (and $|\mathcal{S}|$). (b) Distortion vs. k (and $|\mathcal{S}|$). (c) Number of τ-ghost patterns vs. k (and $|\mathcal{S}|$).



(a) OLD **(b)** OLD **(c)** OLD

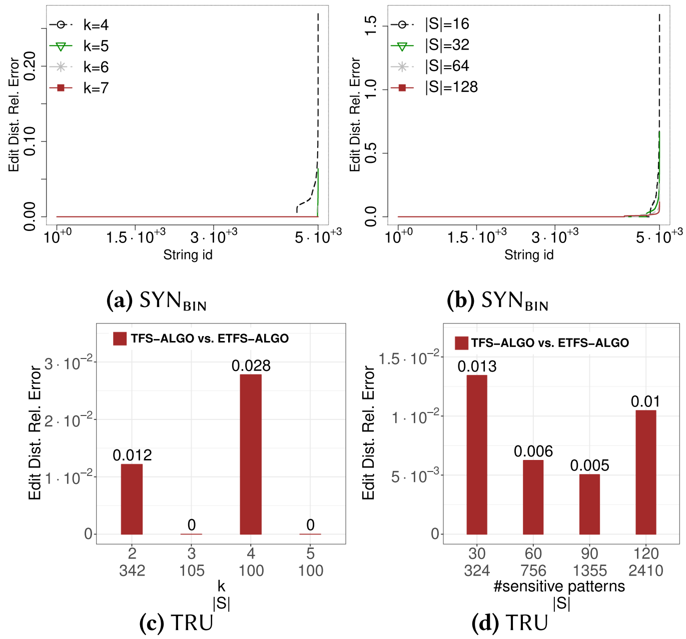
Fig. 13. (a) Distortion, (b) number of τ-ghost patterns, and (c) percentage of implausible patterns vs. ρ.

We now demonstrate that TMI eliminates implausible patterns, while incurring a comparable amount of distortion and ghosts (on average) compared to TM. Specifically, the distortion for TMI was 17% lower than TM on average (see Figure 12(b)), and the number of τ-ghost patterns for TMI was 16.2% lower on average (see Figure 12(c)).

Impact of ρ. We demonstrate that TMI can eliminate implausible patterns, while preserving data utility as well as TM does. This can be seen from Figure 13(a), which shows that the percentage of implausible patterns incurred by TM was 4.1% on average (and up to 5.3%), and from Figures 13(b) and 13(c), which show that TMI caused on average 19.5% lower distortion and 9.4% fewer τ-ghosts, respectively, compared to TM.

8.4 TFS-ALGO vs. ETFS-ALGO

We demonstrate that TFS-ALGO is a very effective heuristic for the ETFS problem. Specifically, it constructs a string X that is either an optimal solution to the problem or it is at slightly larger edit distance from W compared to the exact solution string X_{ED} that is constructed by ETFS-ALGO. This can be seen from Figure 14(a) (resp., Figure 14(b)), which shows that TFS-ALGO constructed optimal solutions (i.e., Edit Distance Relative Error was 0) in 98% (resp., 93%) of the tested strings, on average. These strings are uniformly random and have the same length and alphabet as SYN_{BIN}. Qualitatively similar results were obtained for uniformly random strings of different lengths and alphabet sizes (omitted). In addition, the effectiveness of TFS-ALGO can be seen from Figures 14(c) and 14(d), which show that the Edit Distance Relative Error in TRU was no more than 2.8%. These results are encouraging because, unlike ETFS-ALGO, TFS-ALGO is applicable to large strings such as OLD, MSN, and DNA (recall that its time complexity is linear instead of quadratic in $|W|$).



(a) SYN_{BIN} **(b)** SYN_{BIN}

(c) TRU **(d)** TRU

Fig. 14. Edit Distance Relative Error vs. (a) k (and $|\mathcal{S}|$), and (b) number of sensitive patterns (and $|\mathcal{S}|$) for each of the 50,000 random strings. Edit Distance Relative Error vs. (c) k (and $|\mathcal{S}|$), and (d) number of sensitive patterns (and $|\mathcal{S}|$) for TRU.

9 CONCLUSION

In this article, we introduced the CSD model. The focus of this model is on *guaranteeing* privacy-utility trade-offs in sequential data (e.g., **C1** *vs.* **Π1** and **P2**).

Under this model, we considered two different settings. The common privacy constraint in both settings is that the output string must not contain any sensitive pattern. In the first setting, we aim to generate the minimal-length string that preserves the order of appearance and the frequency of all non-sensitive patterns. We defined a problem, TFS, to capture these requirements, and a variant of it, PFS, that preserves a partial order and the frequency of the non-sensitive patterns but generally produces a shorter string. We developed two time-optimal algorithms, TFS-ALGO and PFS-ALGO, for TFS and PFS, respectively. We also developed MCSR-ALGO, a heuristic that prevents the disclosure of the location of sensitive patterns, ensuring that sensitive patterns are not reinstated, implausible patterns are not introduced, and occurrences of spurious patterns are prevented from the outputs of TFS-ALGO and PFS-ALGO. In the second setting, we aim to generate a string that is at MED from the original string, in addition to preserving the order of appearance and the frequency of all non-sensitive patterns. We defined a problem, ETFS, to capture these requirements, and proposed ETFS-ALGO, an algorithm, which is based on solving specific instances of approximate regular expression matching, to construct such a string.

Our experiments show that string sanitization by TFS-ALGO, PFS-ALGO, and then MCSR-ALGO is both effective and efficient. They also demonstrate that TFS-ALGO can be employed as an effective heuristic to the ETFS problem producing optimal or near-optimal solutions in practice.

We leave the following question unanswered: Given a string X containing #s, a positive integer k, and a positive integer τ, how should we replace the #s in X with letters in Σ, so that the number of distinct length-k τ-ghosts in the resulting string Z is minimized?

ACKNOWLEDGMENTS

We acknowledge the use of the Rosalind HPC cluster hosted by King's College London.

REFERENCES

[1] O. Abul, F. Bonchi, and F. Giannotti. 2010. Hiding sequential and spatiotemporal patterns. *IEEE Transactions on Knowledge and Data Engineering* 22, 12 (2010), 1709–1723.

[2] Osman Abul. 2010. Knowledge hiding in emerging application domains. In *Privacy-Aware Knowledge Discovery: Novel Applications and New Techniques*. CRC Press.

[3] C. C. Aggarwal and P. S. Yu. 2007. On anonymization of string data. In *Proceedings of the 2007 SIAM International Conference on Data Mining*. 419–424.

[4] C. C. Aggarwal and P. S. Yu. 2008. A framework for condensation-based anonymization of string data. *Data Mining and Knowledge Discovery* 16, 3 (2008), 251–275.

[5] C. C. Aggarwal and P. S. Yu. 2008. A general survey of privacy-preserving data mining models and algorithms. In *Privacy-Preserving Data Mining: Models and Algorithms*. Springer.

[6] C. C. Aggarwal and P. S. Yu. 2008. *Privacy-Preserving Data Mining: Models and Algorithms*. Springer.

[7] Yannis Almirantis, Panagiotis Charalampopoulos, Jia Gao, Costas S. Iliopoulos, Manal Mohamed, Solon P. Pissis, and Dimitris Polychronopoulos. 2017. On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for Molecular Biology* 12, 5 (2017).

[8] A. Backurs and P. Indyk. 2015. Edit distance cannot be computed in strongly subquadratic time (Unless SETH is false). In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing*. 51–58.

[9] G. Bernardini, H. Chen, G. Loukides, N. Pisanti, S. P. Pissis, L. Stougie, and M. Sweering. 2020. String sanitization under edit distance. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*. 7:1–7:14.

[10] Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. 2019. String sanitization: A combinatorial approach. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. 627–644.

[11] Giulia Bernardini, Huiping Chen, Gabriele Fici, Grigorios Loukides, and Solon P. Pissis. 2020. Reverse-safe data structures for text indexing. In *Proceedings of the Symposium on Algorithm Engineering and Experiments*. SIAM, 199–213.

[12] F. Bonchi and E. Ferrari. 2010. *Privacy-Aware Knowledge Discovery: Novel Applications and New Techniques*. CRC Press.

[13] L. Bonomi, L. Fan, and H. Jin. 2016. An information-theoretic approach to individual sequential data sanitization. In *Proceedings of the 9th ACM International Conference on Web Search and Data Mining*. 337–346.

[14] L. Bonomi and L. Xiong. 2013. A two-phase algorithm for mining sequential patterns with differential privacy. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*. 269–278.

[15] Volker Brendel, Jacques S. Beckmann, and Edward N. Trifonov. 1986. Linguistics of nucleotide sequences: Morphology and comparison of vocabularies. *Journal of Biomolecular Structure and Dynamics* 4, 1 (1986), 11–21.

[16] Bastien Cazaux, Thierry Lecroq, and Eric Rivals. 2019. Linking indexing data structures to de Bruijn graphs: Construction and update. *Journal of Computer and System Sciences* 104, 1 (2019) 165–183.

[17] R. Chen, G. Acs, and C. Castelluccia. 2012. Differentially private sequential data publication via variable-length N-grams. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 638–649.

[18] G. Cormode, F. Korn, and S. Tirthapura. 2008. Exponentially decayed aggregates on data streams. In *Proceedings of the IEEE 24th International Conference on Data Engineering*. 1379–1381.

[19] M. Crochemore, C. Hancart, and T. Lecroq. 2007. *Algorithms on Strings*. Cambridge University Press.

[20] J. Droppo and A. Acero. 2010. Context dependent phonetic string edit distance for automatic speech recognition. In *Proceedings of the 2010 IEEE International Conference on Acoustics, Speech and Signal Processing*. 4358–4361.

[21] C. Dwork, F. McSherry, K. Nissim, and A. Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Theory of Cryptography Conference*. 265–284.

[22] Sara Foresti. 2011. Microdata protection. In *Encyclopedia of Cryptography and Security, 2nd Ed*, Henk C.A. van Tilborg, and Sushil Jajodia (Eds.). Springer, 781–783.

[23] Benjamin C. M. Fung, Ke Wang, Rui Chen, and Philip S. Yu. 2010. Privacy-preserving data publishing: A survey of recent developments. *ACM Computing Surveys* 42, 4, (June 2010), 53.

[24] J. Gallant, D. Maier, and J. A. Storer. 1980. On finding minimal length superstrings. *Journal of Computer and System Sciences* 20, 1 (1980), 50–58.

[25] A. Gkoulalas-Divanis and G. Loukides. 2011. Revisiting sequential pattern hiding to enhance utility. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1316–1324.

[26] Roberto Grossi, Costas S. Iliopoulos, Robert Mercas, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, and Fatima Vayani. 2016. Circular sequence comparison: Algorithms and applications. *Algorithms for Molecular Biology* 11, 12 (2016).

[27] R. Gwadera, A. Gkoulalas-Divanis, and G. Loukides. 2013. Permutation-based sequential pattern hiding. In *Proceedings of the 2013 IEEE 13th International Conference on Data Mining*. 241–250.

[28] L. Jin, C. Li, and R. Vernica. 2008. SEPIA: Estimating selectivities of approximate string predicates in large Databases. *The VLDB Journal* 17, 5 (Aug 2008), 1213–1229.

[29] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. *The Multiple-Choice Knapsack Problem*. Springer, Berlin, 317–347.

[30] A. Liu, K. Zhengy, L. Liz, G. Liu, L. Zhao, and X. Zhou. 2015. Efficient secure similarity computation on encrypted trajectory data. In *Proceedings of the IEEE International Conference on Data Engineering*. 66–77.

[31] G. Loukides and R. Gwadera. 2015. Optimal event sequence sanitization. In *Proceedings of the 2015 SIAM International Conference on Data Mining*. 775–783.

[32] Grigorios Loukides, Aris Gkoulalas-Divanis, and Bradley Malin. 2010. Anonymization of electronic medical records for validating genome-wide association studies. *Proceedings of the National Academy of Sciences* 107, 17 (2010), 7898–7903.

[33] W. Lu, X. Du, M. Hadjieleftheriou, and B. C. Ooi. 2014. Efficiently supporting edit distance based string similarity search using B^+-trees. *IEEE Transactions on Knowledge and Data Engineering* 26, 12 (2014), 2983–2996.

[34] B. Malin and L. Sweeney. 2000. Determining the identifiability of DNA database entries. In *AMIA*. 537–541.

[35] Christopher D. Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA.

[36] A. Monreale, D. Pedreschi, R. G. Pensa, and F. Pinelli. 2014. Anonymity preserving sequential pattern mining. *Artificial Intelligence and Law* 22, 2 (2014), 141–173.

[37] Eugene W. Myers and Webb Miller. 1989. Approximate matching of regular expressions. *Bulletin of Mathematical Biology* 51, 1 (1989), 5–37.

[38] A. Narayanan and V. Shmatikov. 2008. Robust de-anonymization of large sparse datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. 111–125.

[39] J. Natwichai, X. Li, and M. Orlowska. 2005. Hiding classification rules for data sharing with privacy preservation. In *Data Warehousing and Knowledge Discovery*. Springer, Berlin, 468–477.

[40] D. Pissinger. 1995. A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research* 83, 2 (1995), 394–410.

[41] Solon P. Pissis. 2014. MoTeX-II: Structured MoTif eXtraction from large-scale datasets. *BMC Bioinformatics* 15 (2014), 235.

[42] Mireille Régnier and Mathias Vandenbogaert. 2006. Comparison of statistical significance criteria. *Journal of Bioinformatics and Computational Biology* 4, 2 (2006), 537–552.

[43] P. Samarati and L. Sweeney. 1998. Generalizing data to provide anonymity when disclosing information (abstract). In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 188.

[44] J. Shang, J. Peng, and J. Han. [2016]. MACFP: Maximal approximate consecutive frequent pattern mining under edit distance. In *Proceedings of the 2016 SIAM International Conference on Data Mining*. 558–566.

[45] P. Sinha and A. A. Zoltners. 1979. The multiple-choice knapsack problem.*Operations Research* 27, 3 (1979), 431–627.

[46] X. Sun and P.S. Yu. 2005. A border-based approach for hiding sensitive frequent itemsets. In *Proceedings of the 5th IEEE International Conference on Data Mining*. 426–433.

[47] M. Terrovitis, G. Poulis, N. Mamoulis, and S. Skiadopoulos. 2017. Local suppression and splitting techniques for privacy preserving publication of trajectories. *IEEE Transactions on Knowledge and Data Engineering* 29, 7 (2017), 1466–1479.

[48] George Theodorakopoulos, Reza Shokri, Carmela Troncoso, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. 2014. Prolonging the hide-and-seek game: Optimal trajectory privacy for location-based services. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*. 73–82.

[49] V. S. Verykios, A. K. Elmagarmid, E. Bertino, Y. Saygin, and E. Dasseni. 2004. Association rule hiding. *IEEE Transactions on Knowledge and Data Engineering* 16, 4 (2004), 434–447.

[50] D. Wang, Y. He, E. Rundensteiner, and J. F. Naughton. 2013. Utility-maximizing event stream suppression. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 589–600.

[51] Z. Wen, D. Deng, R. Zhang, and R. Kotagiri. 2019. 2ED: An efficient entity extraction algorithm using two-level edit-distance. In *Proceedings of the IEEE International Conference on Data Engineering*. 998–1009.

[52] Y. Xu, K. Wang, A. W. Fu, and P. S. Yu. 2008. Anonymizing transaction databases for publication. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 767–775.

Received December 2019; revised June 2020; accepted August 2020