Permutation-based Sequential Pattern Hiding

Robert Gwadera
EPFL, Switzerland
robert.gwadera@epfl.ch

Aris Gkoulalas-Divanis
IBM Research, Ireland
arisdiva@ie.ibm.com

Grigorios Loukides
Cardiff University, UK
g.loukides@cs.cf.ac.uk

***Abstract*—Sequence data are increasingly shared to enable mining applications, in various domains such as marketing, telecommunications, and healthcare. This, however, may expose sensitive sequential patterns, which lead to intrusive inferences about individuals or leak confidential information about organizations. This paper presents the first permutation-based approach to prevent this threat. Our approach *hides* sensitive patterns by replacing them with carefully selected permutations that avoid changes in the set of frequent nonsensitive patterns (*side-effects*) and in the ordering information of sequences (*distortion*). By doing so, it retains data utility in sequence mining and tasks based on itemset properties, as permutation preserves the support of items, unlike deletion, which is used in existing works. To realize our approach, we develop an efficient and effective algorithm for generating permutations with minimal side-effects and distortion. This algorithm also avoids implausible symbol orderings that may exist in certain applications. In addition, we propose a method to hide sensitive patterns from a sequence dataset. Extensive experiments verify that our method allows significantly more accurate data analysis than the state-of-the-art approach.**

I. Introduction

Organizations increasingly share sequence data to enable mining applications, including, among others, market basket analysis, web and process modeling, and preference-based services. However, the mining of these data may result in the exposure of *sensitive* sequential patterns, which allow intrusive inferences about individuals, or are confidential to the organization (e.g., they provide competitive edge). This threat has transpired in various domains [1], [6], and it may lead to unsolicited advertising to customers, customer profiling, as well as financial and reputational loss to organizations.

Consider, for example, the data in Fig. 1(a), which an insurance company aims to share with marketing partners. A record in these data contains the order that a customer follows when buying different insurance products a to h (the id column is not released). These data can be mined to find interesting patterns, such as the number of customers who bought sports car, travel, and home insurance in this order (i.e., the records containing the sequential pattern $[f, g, b]$), or the number of customers who bought travel, home, and trailer insurance (i.e., the records containing the itemset $\{g, b, d\}$). However, sharing these data allows the exposure of the sensitive sequential patterns in Fig. 1(b) through *sequential pattern mining* [3]. These patterns are considered exposed (actionable) by the insurance company, when they appear in at least 3 records.

To prevent the exposure of sensitive sequential patterns, the number of records in which these patterns appear (i.e., their *support*) must be lower than a minimum support threshold, which is specified by data owners, based on domain knowledge and/or application requirements. To achieve this, all existing methods [1], [8] *hide* sensitive sequential patterns by deleting symbols from them. To preserve data utility, these methods aim to minimize changes in the set of *frequent nonsensitive* sequential patterns (each such change is termed a *side-effect*) and symbol deletions. Fig. 1(c) shows the result of applying the state-of-the-art method [8] to the data in Fig. 1(a), with minimum support threshold 3. Notice that the method prevented the exposure of the sensitive sequential patterns in Fig. 1(b) by deleting 3 symbols. However, the nonsensitive patterns n_1 to n_5 in Fig. 1(d) are no longer frequent (i.e., 5 side-effects have occurred), because their support is lower than 3.

Clearly, deletion is a very drastic operation, which reduces the support of symbols and often incurs significant utility loss. Consider, for example, the data in Fig. 1(c). Deleting a from record 2 in Fig. 1(a), reduces the support of 6 nonsensitive patterns containing this symbol, including n_1 and n_2 that led to side-effects. Furthermore, deleting d from records 6 and 8 in Fig. 1(a), prohibits 12 (out of 48) sets of insurance products, shown in Fig. 1(e), from being discovered from the data in Fig. 1(c), by applying *frequent itemset mining* [2] with minimum support threshold 3. In addition to frequent itemset mining, deletion may harm data utility for other tasks based on itemset properties, such as association rule mining [2] and clustering using frequent itemsets [7], as well as for tasks in which patterns with both unordered and ordered components need to be discovered (e.g., actionable partial order mining [11]).

Contributions In this paper, we propose the first, permutation-based approach to hiding sensitive sequential patterns. Our approach is founded on the observation that the replacement of a sensitive sequential pattern with any of its permutations in a record can reduce its support and lead to its hiding. This is because a permutation induces a different, non-sensitive[1] ordering of the symbols in the sensitive pattern. For example, the sensitive pattern $s_1 = [a, c, e]$, corresponding to the purchase of insurance for motorbike, jet-ski and then truck, was replaced by $[a, e, c]$ in record 2 of Fig. 1(f). The ordering $[a, e, c]$ is not deemed sensitive by the insurance company, because it does not provide competitive advantage gained by promoting truck insurance to customers who have purchased insurance for motorbike and then jet-ski. The use of permutation can greatly benefit data utility in many domains

[1]Subsequences of sensitive patterns are not considered sensitive in all related works (e.g., [1], [15]) and lifting this assumption is trivial.

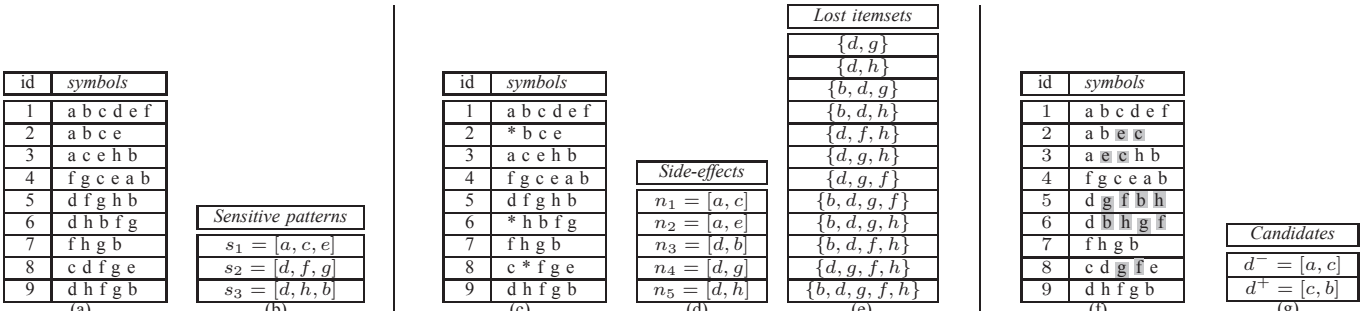


Fig. 1: (a) original dataset S and (b) sensitive patterns $\mathcal{S}$. State-of-the-art method [8]: (c) output (* denotes a deleted symbol), (d) side-effects, and (e) lost frequent itemsets. *PH*: (f) output (swapped symbols are highlighted), and (g) candidate patterns.

where multiple orderings of symbols are possible, as in market basket analysis and preference-based services. This is because, unlike deletion, permutation preserves the support of symbols.

The number of permutations of a sensitive sequential pattern grows factorially with its number of symbols, which provides great choice of permutations. For example, up to 5034 permutations can be used to replace a sensitive pattern comprised of 7 symbols. However, to preserve data utility, the selected permutation needs to (I) avoid side-effects and (II) minimize changes in the ordering information of the record in which the permutation appears (*distortion*). Moreover, a sensitive pattern is typically replaced by many different permutations and in a subset of the records in which it appears. Both the generation of appropriate permutations and the selection of the aforementioned subset of records introduce significant computational challenges. To address these challenges, our work makes the following specific contributions.

First, we investigate the problem of avoiding side-effects, incurred by permutation. We provide tight conditions for identifying patterns that lead to side-effects. These conditions are based on symbol order and support, and they apply to nonsensitive patterns that may become infrequent (*candidates for lost*) or frequent (*candidates for ghost*), after permuting a sensitive pattern. Consider the patterns in Fig. 1(g) and that the pattern $s_1 = [a, c, e]$ in Fig. 1(b) must be hidden using a minimum support threshold 3. d^- is a candidate for lost, whereas d^+ is a candidate for ghost, because the support of d^- (resp., d^+) may become lower than (resp., at least) 3. This occurs, for example, when s_1 is permuted to $[c, e, a]$ in the record 2 of Fig. 1(a), which becomes $[c, b, e, a]$. Given candidates of each type, we prove that generating a permutation that avoids side-effects is NP-complete.

Second, we develop *PDPG*, an algorithm aiming at generating permutations that avoid side-effects and have minimum distortion. *PDPG* employs *Cayley distance* [13] to measure distortion, which quantifies the change of the ordering information of a record, based on the number of symbol swaps. For instance, both permutations $[a, e, c]$ and $[e, a, c]$ of s_1, which transform record 2 in Fig. 1(a) to $[a, b, e, c]$ and $[e, b, a, c]$, respectively, do not cause side-effects with respect to the candidates in Fig. 1(g), but *PDPG* generates $[a, e, c]$, as it has a lower (better) Caley distance. Note that $[a, e, c]$ incurs a smaller change of the ordering information of record 2, because it is produced by a single swap of c and e in s_1, whereas two swaps are required to produce $[e, a, c]$. Furthermore, *PDPG* is coupled with a heuristic for generating permutations with minimal number of side-effects, when no permutation that avoids all side-effects can be constructed. Moreover, this algorithm does not generate permutations containing implausible symbol orderings (e.g., a pattern representing the addition of an extra driver to a car insurance policy before the purchase of the policy). Thus, data utility is preserved and the identification of such orderings as artifacts of sanitization is prevented.

Third, we propose *PH*, a novel sequential pattern hiding algorithm that works as follows: (I) it selects records with a small upper-bound on the number of side-effects that can be incurred by their *sanitization* (i.e., the hiding of all sensitive patterns appearing in the records), and (II) it sanitizes each selected record, by replacing the sensitive patterns in the record with appropriate permutations, generated by *PDPG*. For example, when applied to Fig. 1(a) with minimum support threshold 3, *PH* produced the data in Fig. 1(f). As no side-effects were incurred and no symbols were deleted, the data produced by *PH* remain as useful as the original data, for sequential pattern mining and tasks based on itemset properties, as opposed to the data in Fig. 1(c).

Fourth, we experimentally demonstrate that our approach permits highly accurate sequential pattern mining, frequent itemset mining, and estimation of the support of items and frequent itemsets, which is important in several data mining tasks. Specifically, *PH* significantly outperformed the state-of-the-art method [8] in terms of data utility, as it incurred at least 21% fewer side-effects, preserved at least 77% more frequent itemsets, and was many times more effective at preserving the support of items and frequent itemsets.

Paper organization Section II presents the background, Section III our permutation-based methodology, and Section IV our sanitization algorithm. Sections V, VI, and VII present related work, our experimental evaluation, and a discussion on attacks based on background knowledge, respectively. Section VIII concludes the paper.

II. Background

In this section, we present necessary concepts to explain our approach and formulate the problem statement.

Preliminaries Let $\mathcal{A} = \{a_1, a_2, \ldots, a_{|\mathcal{A}|}\}$ be an alphabet of size $|\mathcal{A}|$ and $\mathbf{S} = \{t^{(1)}, t^{(2)}, \ldots, t^{(|\mathbf{S}|)}\}$ be a collection of sequences with cardinality $|\mathbf{S}|$. $\mathbf{S}$ will also be referred to as

the *original dataset* and its elements as *transactions*. The i-th transaction in $\mathbf{S}$ is denoted with $t^{(i)} = [\mathrm{s}_1^{(i)}, \mathrm{s}_2^{(i)}, \ldots, \mathrm{s}_{|t^{(i)}|}^{(i)}]$, where $\mathrm{s}_l^{(i)} \in \mathcal{A}$, $l \in [1, |t^{(i)}|]$. A transaction $t^{(i)}$ contains symbols that are not necessarily distinct, its *length* (i.e., number of symbols) is denoted with $|t^{(i)}|$, and it has an id $t^{(i)}.id = i$. A sequence $s = [\mathrm{s}_1, \mathrm{s}_2, \ldots, \mathrm{s}_{|s|}]$ has length $|s|$ and is a *subsequence* of sequence $s' = [\mathrm{s}'_1, \mathrm{s}'_2, \ldots, \mathrm{s}'_{|s'|}]$, denoted $s \sqsubseteq s'$, if there exist integers $1 \leq i_1 \leq i_2 \ldots \leq i_{|s|}$ such that $\mathrm{s}_1 = \mathrm{s}'_{i_1}, \mathrm{s}_2 = \mathrm{s}'_{i_2}, \ldots, \mathrm{s}_{|s|} = \mathrm{s}'_{i_{|s|}}$. We also write that s' is a *supersequence* of s and that s is *contained* in s'. The set of common symbols of s and s' is denoted with $s \widehat{\cap} s'$.

The *support* of a sequence s in $\mathbf{S}$, denoted by $sup_{\mathbf{S}}(s)$, is defined as the number of transactions in $\mathbf{S}$ that contain s as a subsequence. Given a support threshold $minSup$, a sequence s is a *frequent sequential pattern*, if $sup_{\mathbf{S}}(s) \geq minSup$, and *infrequent* otherwise. The problem of sequential pattern mining [3] is to find the set of all frequent sequential patterns in $\mathbf{S}$, given $minSup$. This set is denoted by $\mathcal{F}_{\mathbf{S},minSup}$, and we may omit $\mathbf{S}$ and/or $minSup$, when it is clear from the context. The support has an *anti-monotonic* property, i.e., $sup_{\mathbf{S}}(s) \geq sup_{\mathbf{S}}(s')$, if $s \sqsubseteq s'$.

A permutation $\pi(\mathcal{E})$ of a set of elements $\mathcal{E} = \{0, \ldots, n-1\}$ is a bijection (1-1 mapping) from $\mathcal{E}$ to itself. In this paper, ***$\mathcal{E}$ refers to positions of symbols in sequences***, and not to the actual symbols of $\mathcal{A}$, unless stated otherwise. Given a sequential pattern s, $\pi(s)$ refers to a permutation of positions of s. For simplicity, we may use symbols of s that coincide with their positions.

A permutation π is represented in *two-line notation* as

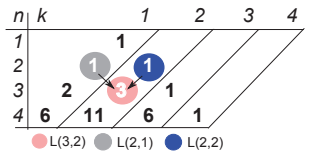
$$\pi = \begin{pmatrix} 0 & 1 & \ldots & n-1 \\ \pi(0) & \pi(1) & \ldots & \pi(n-1) \end{pmatrix}, \quad (1)$$

where the mapping is $\{i \rightarrow \pi(i)\}$, $\forall i \in \mathcal{E}$. The permutation that *fixes* all elements (i.e., maps each $i \in \mathcal{E}$ to itself) is the *identity permutation* $\mathcal{I}$. The product of permutations π and σ is given by their *composition* $\rho = \sigma\pi$, $\rho(i) = \pi(\sigma(i))$. So, the permutations multiply in the order in which they are written. The *inverse permutation* π^{-1} of π is a permutation such that $\pi\pi^{-1} = \pi^{-1}\pi = \mathcal{I}$.

Given a permutation $\pi(s)$, we can group the elements in s and $\pi(s)$ that trade their places (swap) in Equation (1) to obtain the *cycle notation* of $\pi(s)$. Figure 2(a) presents the cycle notation of all permutations of $s = [0, 1, 2]$.

$\pi(s)$	cycle notation	# swaps to recover s
[0, 1, 2]	[(0), (1), (2)]	0
[0, 2, 1]	[(0), (1, 2)]	1
[1, 0, 2]	[(0, 1), (2)]	1
[1, 2, 0]	[(0, 1, 2)]	2
[2, 0, 1]	[(0, 2, 1)]	2
[2, 1, 0]	[(0, 2), (1)]	1

(a)



(b)

Fig. 2: (a) Permutations $\pi(s)$ of $s = [0, 1, 2]$, their corresponding cycle notation, and the number of swaps needed to recover s. (b) The triangle of Stirling's cycle numbers for $n = 4$.

Given $x_1, \ldots, x_n$ distinct elements of $\mathcal{E}$, a *cycle* $(x_1, \ldots, x_n)$ is a permutation that maps x_1 to x_2, x_2 to x_3, ..., x_n to x_1, while fixing any other element in $\mathcal{E}$. A cycle $(x_1, \ldots, x_n)$ has length n and is a product of $n-1$ *transpositions* (i.e., cycles of length 2) [13], i.e.,

$$(x_1, x_2, \ldots, x_n) = (x_1, x_2)(x_1, x_3) \ldots (x_1, x_n). \quad (2)$$

For example, the cycle $(0, 1, 2)$ corresponds to $[1, 2, 0]$ and is the product of transpositions $(0, 1)(0, 2)$. In fact, applying the transposition $(0, 1)$ to $s = [0, 1, 2]$, changes s to $[1, 0, 2]$, and applying $(0, 2)$ to $[1, 0, 2]$, changes s to $[1, 2, 0]$.

The *Cayley distance* between two permutations ρ and π, denoted with $\mathcal{C}(\rho, \pi)$, is defined as the minimum number of transpositions required to change ρ to π by multiplication [13]

$$\mathcal{C}(\rho, \pi) = min\{n | \rho\tau_1 \ldots \tau_n = \pi\}, \quad (3)$$

where τ_i is a transposition (swap). Thus, $\mathcal{C}(\pi, \mathcal{I})$ denotes the Cayley distance between π and $\mathcal{I}$, the identity permutation of the first argument. In particular, it holds that

$$\mathcal{C}(\pi, \mathcal{I}) = n - c(\pi), \quad (4)$$

where $c(\pi)$ is the number of cycles in π and n is the number of symbols in π. For example, $C([0, 2, 1], \mathcal{I}) = 3 - 2 = 1$, as $[0, 2, 1]$ has 2 cycles.The number of permutations of n symbols that have k cycles is equal to the Stirling cycle number [17]:

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} \quad (5)$$

k-Cycle permutation generation The generation of all *k-cycle permutations* (i.e., permutations with k cycles) of a sequence with n symbols can be performed by the *East-side/west-side* algorithm [20]. This algorithm recursively constructs k-cycle permutations by following a triangle similar to that in Figure 2(b), for the given n and k, in a bottom-up manner. Let $L(n, k)$ be the set of k-cycle permutations of n symbols, which is split into two subsets; one with permutations in which the symbol n is alone in a cycle (*east-side*), and another with permutations in which n shares a cycle with other symbols (*west-side*). To generate the east-side subset, the algorithm takes the set $L(n-1, k-1)$ and inserts, into each permutation, a cycle that contains only symbol n. To generate the west-side subset, it takes $L(n-1, k)$ and, for each permutation π in this set, it constructs $n-1$ new permutations. This is performed by inserting the symbol n successively, between consecutive elements in each of the cycles in π. For example, to construct $L(3, 2)$, the algorithm takes $L(2, 1) = \{[(0, 1)]\}$ and creates a new cycle, which contains only 2, to obtain $[(0, 1), (2)]$ (east-side element). Then, it takes $L(2, 2) = \{[(0), (1)]\}$ and inserts the symbol 2 into each position between consecutive symbols in $[(0), (1)]$ to obtain $[(0, 2), (1)]$ and $[(0), (1, 2)]$ (west-side elements).

The generation of a *random* k-cycle permutation, can be performed by the *Random-permutation East-side/west-side* algorithm [20]. This algorithm constructs $L(n, k)$ by first selecting randomly an east-side or west-side permutation. The probability of choosing the east-side element is $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix} / \begin{bmatrix} n \\ k \end{bmatrix}$. If an east-side element is selected, then it recursively chooses a random permutation from $L(n-1, k-1)$ and inserts a new cycle, which contains only symbol n, into it. Otherwise, it recursively selects a random permutation from $L(n-1, k)$ and inserts n into a randomly chosen position, between consecutive elements in $L(n-1, k)$. This algorithm needs $O(n^2)$ time.

In the following, we define the problem we aim to solve.

Problem (Sequential pattern hiding) *Given the original dataset* $\mathbf{S}$*, a support threshold* $minSup$ *and a set of sensitive sequential patterns* $\mathcal{S}$ *(selected by data*

owners), construct a version $\mathbf{S}'$ *of* $\mathbf{S}$ *such that:* (I) $sup_{\mathbf{S}'}(s) < minSup$*, for each* $s \in \mathcal{S}$*,* (II) $\mathcal{F}_{\mathbf{S}'} = \mathcal{F}_{\mathbf{S}} - \mathcal{S}^*$*, where* $\mathcal{S}^* = \{s^* \in \mathcal{F}_{\mathbf{S}'} | s \sqsubseteq s^*, s \in \mathcal{S}\}$*, and* (III) $\mathbf{S}'$ *is constructed with minimum distortion*[2].

The problem requires sanitizing the original dataset $\mathbf{S}$ so that (I) no sensitive sequential pattern $s \in \mathcal{S}$ can be mined from the released dataset $\mathbf{S}'$ at a support threshold $minSup$ or higher; (II) no side-effects are introduced to $\mathbf{S}'$ in terms of (i) sequential patterns in $\mathcal{F}_{\mathbf{S}} - \mathcal{S}^*$ that are infrequent in $\mathbf{S}'$ (*lost*) or (ii) infrequent sequential patterns in $\mathbf{S}$ that are frequent in $\mathbf{S}'$ (*ghost*), and (III) $\mathbf{S}'$ and $\mathbf{S}$ are as "similar" as possible (the notion of distortion will be discussed later). Unfortunately, the problem is NP-hard (the proof follows from [1]). Also, note that we consider a single support threshold $minSup$. Extensions for multiple support thresholds are straightforward and can be implemented following [1].

The three goals in our problem are not equally important. That is, a sequential pattern hiding method must protect all sensitive sequential patterns, but it should prioritize the minimization of the number of side-effects over that of distortion [8]. This is because it is essential that the released dataset enables the discovery of nonsensitive frequent patterns.

III. Hiding methodology

This section presents our *PDPG* algorithm and a heuristic for increasing its effectiveness. Subsequently, it explains how this algorithm deals with *forbidden patterns* (e.g., implausible symbol orderings). *PDPG* takes as input candidate patterns for lost and ghost, which are identified as discussed below.

Identifying candidates for ghost and lost Let s be a sensitive pattern and $\Pi(s)$ be the set of all its permutations. Observe that the replacement of s with any of the $|s|!$ permutations in $\Pi(s)$, except the identity permutation, in a transaction, can lead to reducing the support of s by 1. Thus, to hide the sensitive pattern s from the original dataset $\mathbf{S}$, it suffices to perform such replacements in $sup_{\mathbf{S}}(s) - (minSup - 1)$ transactions. However, replacing s by a permutation may decrease or increase the support of nonsensitive sequential patterns in $\mathbf{S}$, incurring lost or ghost patterns, respectively. This can be prevented by identifying permutations that avoid side-effects and using them to replace s.

We now provide detailed conditions for the decrease or increase of the support of a nonsensitive pattern s', when a sensitive pattern s in a transaction t is replaced with its permutation $\pi(s)$. The first type of conditions we consider are *symbol-based* and explain the change in the support of s', based on the interplay of symbols in s and s', as follows:

- S1 When $s \cap s' \neq \varnothing$ (i.e., s and s' share symbols), the support of s' may increase or decrease.
- S2 When the conditions (a)-(c) below hold simultaneously, the support of s' may decrease:
  - a) s and s' have a common subsequence s''
  - b) s and s' co-occur in t, by sharing s''
  - c) the permutation of s causes at least one of symbol in s'' to change its relative ordering with respect to other symbols in s''. This requires one or more swaps between: (i) symbols in s'', or (ii) a symbol in $s \setminus s''$ and another in s''.
- S3 When the conditions (a)-(c) below hold simultaneously, the support of s' may increase:
  - a) s and s' have a common set of symbols $s^* = s \cap s'$, but s' is not a subsequence of s, and s is not a subsequence of s'
  - b) s co-occurs in the transaction t with a permutation $\pi(s')$ of s', by sharing s^*
  - c) the permutation of s reverses $\pi(s')$ and recovers s'. The reversal of $\pi(s')$ requires one or more swaps between: (i) symbols in s^*, or (ii) a symbol in $s \setminus s^*$ and another in s^*.

There are two special cases for s'' in S2: (i) $s'' = s'$, when $s' \sqsubseteq s$, and (ii) $s'' = s$, when $s \sqsubseteq s'$. Similarly, the special cases for s^* in S3 are: (i) $s^* = s'$, and (ii) $s^* = s$. Examples for conditions S2 and S3 are shown in Figs. 3(a) and 3(b).

Swapping symbols in	s	s'	s''	t	$\pi(s)$	t after $\pi(s)$
s''	$[a, b, c]$	$[a, d, c]$	$[a, c]$	[a , b, d, c]	$[c, b, a]$	[c , b, d, a]
$s \setminus s''$, s''	$[a, b, c]$	$[d, b, c]$	$[b, c]$	[a , d, b, c]	$[c, b, a]$	[c , d, b, a]
$s \setminus s''$, s''	$[b, a]$	$[d, c, a]$	$[a]$	[d, b , c, a]	$[a, b]$	[d, a , c, b]

(a)

Swapping symbols in	s	s'	s^*	t	$\pi(s)$	t after $\pi(s)$
s^*	$[a, b, c]$	$[c, d, a]$	$\{a, c\}$	[a , b, d, c]	$[c, b, a]$	[c , b, d, a]
$s \setminus s^*$, s^*	$[a, b, c]$	$[b, d, c]$	$\{b, c\}$	[a , d, b , c]	$[b, a, c]$	[b , d, a , c]
$s \setminus s^*$, s^*	$[a, b, c]$	$[a, d, b]$	$\{a, b\}$	[a, b , d, c]	$[a, c, b]$	[a, c , d, b]

(b)

Fig. 3: Permuting s to $\pi(s)$ in a transaction t can (a) decrease or (b) increase the support of s'. Swapped symbols are highlighted.

We also show that, whether or not a nonsensitive pattern s' can lead to a side-effect, depends on its support, before permuting s, and on the number of transactions to which permutation is applied, denoted with $\Delta_{supp}(s|\pi(s))$. Combining this observation with conditions S2 and S3, leads to the following conditions, which are used to identify candidates for lost and ghost.

- C1 A frequent, nonsensitive pattern s' may become *infrequent* (lost), as a result of permuting s in $\Delta_{supp}(s|\pi(s))$ transactions in $\mathbf{S}$, if it satisfies S1 and S2, and its support, before permuting s, satisfies
$$sup_{\mathbf{S}}(s') < minSup + \Delta_{supp}(s|\pi(s)). \quad (6)$$
- C2 An infrequent, nonsensitive pattern s' may become *frequent* (ghost), as a result of permuting s in $\Delta_{supp}(s|\pi(s))$ transactions in $\mathbf{S}$, if it satisfies S1 and S3, and its support, before permuting s, satisfies
$$sup_{\mathbf{S}}(s') \geq minSup - \Delta_{supp}(s|\pi(s)). \quad (7)$$

Observe that C1 models a worst case, in which s' satisfies S1 and S2, and it co-occurs with s, in each of the $\Delta_{supp}(s|\pi(s))$ transactions. In this case, the support of s' in $\mathbf{S}$ decreases by $\Delta_{supp}(s|\pi(s))$, and, for s' to become infrequent, we need $sup_{\mathbf{S}}(s') - \Delta_{supp}(s|\pi(s)) < minSup$, from which we get (6). A similar logic is applied to derive C2.

Conditions C1 and C2 can significantly improve the efficiency of identifying candidates for lost or ghost, as, in practice, they apply to a small fraction of the nonsensitive patterns. To employ these conditions, we first use Equations (6) and (7), which establish bounds on the support of patterns, then

[2]Following [8], the loss of a sequence s^* in $\mathcal{F}_{\mathbf{S}}$ that is a supersequence of a pattern $s \in \mathcal{S}$ is not considered as side-effect, as s^* will be inevitably hidden when we hide s.

apply S1 to prune pairs of s and s' that do not share symbols, and last check S2 or S3. This heuristic order is effective at minimizing the number of checked patterns.

Checking conditions S1 and S2 is straightforward, because s' is a subsequence of the transaction t. On the other hand, the checking of S3 requires taking into account the transaction and the potentially many ways to recover s' (see S3(c)). To check whether s' can be recovered by permuting s, we develop a simple algorithm, called *Restricted Subsequence of a Permutation (RSP)*. *RSP* matches s' to the symbols in t in order to identify whether s' can occur in this transaction, after permuting s. *RSP* needs $O(|s'|)$ time and space, and it is illustrated in the following example.

Example 1 *Consider the patterns s and s', and the transaction $t = [a, b, d, c]$ in the first row of Fig. 3(b).* RSP *replaces the symbols of $s = [a, b, c]$ in t with placeholders $\#_i$, for $i = 0, 1, 2$, to obtain $[\#_0, \#_1, d, \#_2]$, and then performs a pattern matching of $s' = [c, d, a]$ in $[\#_0, \#_1, d, \#_2]$. Let $\Sigma = \{a, b, c\}$ be the set of symbols from s that must be matched in $[\#_0, \#_1, d, \#_2]$ and V a vector holding intermediate matching results. Initially, every placeholder points to the same set of symbols (i.e., $\#_i = \Sigma$, for $i = 0, 1, 2$), as it can be assigned to any symbol in Σ. Next,* RSP *considers each symbol in s' in the order of appearance. First, it considers c, which is matched to $V[0] = \#_0$ and subtracted from Σ. Then,* RSP *considers d. Thus, it skips $\#_1$ and matches $V[1] = d$. In this case, Σ remains unchanged. Last,* RSP *considers a, which is matched to $V[2] = \#_2$ and subtracted from Σ. This results in $V = [c, d, a]$, $\#_1 = b$, so s' may appear in t and satisfies S3.*

Generating pattern-constrained permutations with minimal Cayley distance Before discussing how permutations that avoid side-effects and minimize Cayley distance can be constructed, we define the notion of *pattern-constrained* permutation. Let s be a sensitive sequential pattern, t be a transaction in $\mathbf{S}$, and $\mathcal{D}^{-(t)}(s)$ and $\mathcal{D}^{+(t)}(s)$ be sets of candidate patterns to lose and gain occurrence in t, respectively, as a result of replacing s with certain of its permutations. The permutation $\pi(s)$ is *pattern-constrained*, if all patterns in $\mathcal{D}^{-(t)}(s)$ and in $\mathcal{D}^{+(t)}(s)$ do not lose and gain occurrence in t, respectively, when $\pi(s)$ replaces s. Generating a pattern-constrained permutation is NP-complete, as proven below.

Theorem 1 *Given a sensitive sequential pattern s, a transaction $t \in \mathbf{S}$, a set $\mathcal{D}^{-(t)}(s)$ of candidates to lose occurrence and a set $\mathcal{D}^{+(t)}(s)$ of candidates to gain occurrence in t, finding a pattern-constrained permutation $\pi(s)$ of s is NP-complete.*

Proof: (Sketch) The problem can be reduced from the NP-complete SAT problem. The reduction involves: (I) constructing a Boolean formula $\mathcal{B}$ in CNF, whose literals correspond to different ways in which patterns in $\mathcal{D}^{-(t)}(s)$ and $\mathcal{D}^{+(t)}(s)$, as well as s, can appear in t, and (II) showing that $\mathcal{B}$ is true if and only if $\pi(s)$ solves our problem. ■

Furthermore, we define the notion of *partial* permutation, which is central to our permutation generation algorithm. Given a pattern s, a partial permutation of s, $\pi(s|m)$, is a permutation of a prefix of positions of s, from 0 (start of s) to $m-1$, for $m \leq |s|$. Given a transaction t and a pattern $d^{-(t)}(s) \in \mathcal{D}^{-(t)}(s)$ (respectively, $d^{+(t)}(s) \in \mathcal{D}^{+(t)}(s)$), a partial permutation $\pi(s|m)$ of s, where $m \leq |s|$, is:

R1 *satisfiable* w.r.t. $d^{-(t)}(s)$ if
a) $\mathcal{C}(\pi(s|m)(d^{-(t)}(s)), \mathcal{I}) \in (0, n-m]$, or
b) $\mathcal{C}(\pi(s|m)(d^{-(t)}(s)), \mathcal{I}) = 0$ (perfect ordering)

R2 *satisfiable* w.r.t. $d^{+(t)}(s)$ if
a) $\mathcal{C}(\pi(s|m)(d^{+(t)}(s)), \mathcal{I}) = 0$ **and** $n - m > 0$, or
b) $\mathcal{C}(\pi(s|m)(d^{+(t)}(s)), \mathcal{I}) > 0$ (disorder)

where $\pi(s|m)(d^{-(t)}(s))$ and $\pi(s|m)(d^{+(t)}(s))$ denotes the result of applying $\pi(s|m)$ to transaction t and then projecting on $d^{-(t)}(s)$ and $d^{+(t)}(s)$, respectively.

R1 states that the occurrence of $d^{-(t)}(s)$ can be preserved, by applying swaps involving the remaining symbols of s, if there are such symbols (i.e., $n - m > 0$). Furthermore, according to R2, if there are remaining symbols, the occurrence of $d^{+(t)}(s)$ may be avoided, since a single swap suffices for that. Moreover, by comparing R1 and R2, it can be seen that, preventing a pattern $d^{+(t)}(s)$ from gaining an occurrence in t and becoming ghost, is much easier than preventing a pattern $d^{-(t)}(s)$ from losing an occurrence in t and becoming lost.

The following example illustrates a satisfiable permutation w.r.t. a pattern $d^{-(t)}(s)$ and a pattern $d^{+(t)}(s)$.

Example 2 *Consider the transaction $t = [c, d, f, g, e]$, the sensitive sequential pattern $s = [d, f, g]$, and the candidate patterns $d^{-(t)}(s) = [c, d, f]$ and $d^{+(t)}(s) = [f, d, e]$ to lose and gain an occurrence in t, respectively. By referring to symbol positions, we can write $t = [0, 1, 2, 3, 4]$, $s = [1, 2, 3]$, $d^{-(t)}(s) = [0, 1, 2]$, and $d^{+(t)}(s) = [2, 1, 4]$. Also, let $\pi(s|2) = [2, 1]$ be a partial permutation of s. Observe that*

- *$\pi(s|2)(d^{-(t)}(s)) = [0, 2, 1]$, $\mathcal{C}(\pi(s|2)(d^{-(t)}(s)), \mathcal{I}) \in (0, 1]$. Thus, $\pi(s|2)$ is satisfiable w.r.t. $d^{-(t)}(s)$, according to R1(a).*
- *$\pi(s|2)(d^{+(t)}(s)) = [2, 1, 4]$, $\mathcal{C}(\pi(s|2)(d^{+(t)}(s)), \mathcal{I}) = 0$ and $n - m = 1 > 0$. Thus, $\pi(s|2)$ is satisfiable w.r.t. $d^{+(t)}(s)$, according to R2(a).*

In fact, s becomes $[2, 1, 3]$ after applying $\pi(s|2)$, and, by applying the swap $(2, 3)$ to s, we obtain $s = [3, 1, 2]$ and $t = [0, 3, 1, 2, 4]$. This recovers the occurrence of $d^{-(t)}(s)$ and prevents the occurrence of $d^{+(t)}(s)$ in t.

Note that conditions R1 and R2 assume a single pattern $d^{-(t)}(s)$ and $d^{+(t)}(s)$, respectively. However, there can be a set of candidates $\mathcal{D}^{-(t)}(s)$ to lose occurrence in a transaction t, and a set of candidates $\mathcal{D}^{+(t)}(s)$ to gain occurrence in t. When there are multiple patterns in $\mathcal{D}^{-(t)}(s)$ (resp., $\mathcal{D}^{+(t)}(s)$), we apply condition R1 (resp., R2) to each of these patterns, and use the maximum of the Caley distance, over the patterns, to determine if $\pi(s|m)$ is satisfiable or not.

To tackle our problem, we must generate pattern-constrained permutations that hide all sensitive patterns, from each transaction of $\mathbf{S}$ that needs to be sanitized, and incur minimum distortion. Clearly, the generation of such permutations remains NP-complete. Furthermore, applying *Random-permutation East-side/west-side* (see Section II) and then

checking if the generated permutation is satisfiable, is prohibitively expensive, due to the extremely large number of random trials required.

Thus, we develop the *Pattern-constrained Distance-based Permutation Generation* (*PDPG*) algorithm. Given a transaction t, containing a sensitive pattern s, *PDPG* aims at generating a random, partial permutation $\pi(s|m)$ that is pattern-constrained and has minimum Cayley distance (i.e., maximum number of cycles k). The following steps describe *PDPG*.

1) Let $\pi(1|1)$ be the permutation $[(0)]$ in cycle notation.
2) For $m = 2, \ldots, n$ and for $k = m, \ldots, 1$, execute steps 2 to 6.
3) Apply *Conditional Random Search* to choose the type of $\pi(s|m)$ in terms of cycle structure.
4) If $\pi(s|m)$ is of west-side type, then
   i Generate a set of positions P leading to satisfiable *extensions* of $\pi(s|m)$.
   ii Randomly select a position from P.
   iii Extend $\pi(s|m)$ using the selected position.
5) If $\pi(s|m)$ is of east-side type, then extend $\pi(s|m)$ using the next available position for a new cycle.
6) If $m = n$, then return $\pi(s|m)$, which is satisfiable and has the minimum Cayley distance.

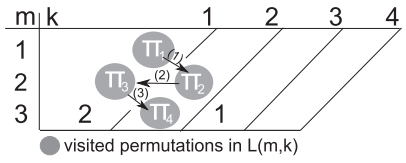
PDPG uses a function *Conditional Random Search* to guide the random search in the space of possible solutions, while trying to generate $\pi(s|m)$ (step 3). This function first computes all possible *extensions* of $\pi(s|m)$ (i.e., each random partial permutation $\pi(s|m+1)$ that can be obtained from $\pi(s|m)$) for the current m and k, and, based on these extensions, it computes the distribution of satisfiable extensions from east-side and west-side permutations. This is required to compute the probability of selecting an east-side or west-side permutation, which is not known in advance. If $\pi(s|m)$ is of west-side type, then *PDPG*: (I) constructs a set P, containing all the positions that will lead to satisfiable extensions of $\pi(s|m)$, (II) randomly selects a position in P, and (III) uses this position to extend $\pi(s|m)$, as in the *Random-permutation East-side/west-side* algorithm (step 4). If $\pi(s|m)$ is of east-side type, the algorithm extends $\pi(s|m)$, using the next available position for creating a new cycle (step 5). Last, when $m = n$, *PDPG* returns a satisfiable permutation $\pi(s|m)$ with the minimum Cayley distance.

Fig. 4 shows how *PDPG* generates a permutation, for a transaction $t = [0, 1, 2, 3, 4]$, sensitive pattern $s = [1, 2, 3]$, and candidate patterns $d^{-(t)}(s) = [2, 3, 4]$ and $d^{+(t)}(s) = [3, 2, 4]$. The algorithm follows the triangle in Fig. 4(b) and generates a randomly selected, satisfiable $\pi(s|m)$ at each element of the triangle (i.e., $L(m, k)$). That is, *PDPG* starts by $\pi_1 = \pi(1|1)$ and extends it to $\pi_2 = \pi(2|2)$, which is of east-side type. The extension is performed using position 1, which is the next available position for a cycle. Then, *PDPG* generates $\pi_3 = \pi(2|1)$, which is of west-side type by extending it using position 1. Last, π_4 is constructed and returned for $m = n = 3$ and $k = 2$. Note that, replacing s with π_4, transforms t to $[0, 2, 1, 3, 4]$. Thus, $d^{-(t)}(s) = [2, 3, 4]$ (resp., $d^{+(t)}(s) = [3, 2, 4]$) does not lose (resp., gain) occurrence in t.

However, it may not be possible for *PDPG* to satisfy all constraints, due to the NP-completeness of the problem. In this case, we employ a heuristic that aims to generate a permutation

m	k	$\pi(s\|m)$	constraints			
			$d^{-(t)}(s) = [2,3,4]$		$d^{+(t)}(s) = [3,2,4]$	
			Satisf.	$\mathcal{C}(d^{-(t)}(s), \mathcal{I})$	Satisf.	$\mathcal{C}(d^{+(t)}(s), \mathcal{I})$
1	1	$\pi_1 = [(0)]$	true	0	true	1
2	2	$\pi_2 = [(0),(1)]$	true	0	true	1
2	1	$\pi_3 = [(0,1)]$	true	0	true	1
3	2	$\pi_4 = [(0,1),2]$	true	0	true	1

(a)



(b)

Fig. 4: (a) Generating a satisfiable partial permutation, and (b) the steps on the triangle.

satisfying the maximum possible number of constraints. The heuristic, called *Rank Reduce PDPG* (*RR-PDPG*), ranks the constraints, based on their chance of being satisfied, and then progressively reduces the number of considered constraints given as input to *PDPG*. Our heuristic is based on the following observations:

- The shortest patterns in $\mathcal{D}^{-(t)}(s)$ are the easiest to be preserved, since they require fewer positions to have a particular ordering. Therefore, we rank the patterns in $\mathcal{D}^{-(t)}(s)$ in increasing order of length.
- The longest patterns in $\mathcal{D}^{+(t)}(s)$ are the hardest to be created, since they require more symbols to follow a particular ordering. Therefore, we rank the patterns in $\mathcal{D}^{+(t)}(s)$ in decreasing order of length.
- The patterns in $\mathcal{D}^{+(t)}(s)$ are easier to be preserved than those in $\mathcal{D}^{-(t)}(s)$. Therefore, we first attempt to satisfy the constraints for $\mathcal{D}^{+(t)}(s)$.

The following steps describe *RR-PDPG*.

1) Rank the patterns in $\mathcal{D}^{-(t)}(s)$, in increasing order of length, and the patterns in $\mathcal{D}^{+(t)}(s)$, in decreasing order of length.
2) Let v^- and v^+ be vectors containing all patterns in $\mathcal{D}^{-(t)}(s)$ and $\mathcal{D}^{+(t)}(s)$, respectively. The vector v^- is sorted increasingly w.r.t. the length of patterns in $\mathcal{D}^{-(t)}(s)$, whereas the vector v^+ is sorted decreasingly w.r.t. the length of patterns in $\mathcal{D}^{+(t)}$.
3) Let variables $n^- = |v^-|$ and $n^+ = |v^+|$ store the ranges in v^- and v^+ respectively.
4) Recursively select the n^+ first elements in v^+ and try to find a permutation satisfying these elements using *PDPG*. If this fails, set $n^+ = \frac{n^+}{2}$ and execute step 4 again. Else, continue to step 5.
5) Recursively select the n^- first elements in v^- and try to find a permutation satisfying n^- and the previously selected n^+ elements using *PDPG*. If this fails, set $n^- = \frac{n^-}{2}$ and execute step 5 again. Else, return.

After ranking the patterns, *RR-PDPG* tries to find a large subset of candidates for ghost, which are satisfied, by recursively selecting the first half of elements in v^+ (steps 1-4). Then, it attempts to add a subset of candidates for lost to the subset already found, by recursively selecting the first half of elements in v^- (step 5). Steps 4 and 5 are executed $O(\log(M))$ times, where $M = \max\{|\mathcal{D}^{-(t)}(s)|, |\mathcal{D}^{+(t)}(s)|\}$,

and an unconstrained permutation with minimum Caley distance is generated, in the worst-case when $n^+ = n^- = 0$.

The worst case time complexity of *PDPG* is $O(n^3M)$ (vs. $O(n^2)$ for the unconstrained case), where n is the length of the sensitive pattern. This is because $O(n^2)$ is needed to follow the triangle, $O(n)$ time to construct the extensions, and M to check their satisfiability. Thus, *RR-PDPG* needs $O(n^3 M \log(M))$, in the worst case. Since n is a small constant in practice, both *PDPG* and *RR-PDPG* are very efficient.

Dealing with forbidden patterns In certain application domains, such as process mining, there may be implausible symbol orderings. Such orderings, henceforth referred to as *forbidden patterns*, must be prevented from occurring in the sanitized data to help data utility and ensure that attackers do not use them to distinguish between original and sanitized transactions. To achieve this, forbidden patterns are given as input to *PDPG* and treated similarly to candidates for ghost, with the difference that they are checked using conditions S1 and S3 but not C2. Thus, *PDPG* generates *only permutations that contradict all forbidden patterns*, and, when this is not possible, it does not invoke *RR-PDPG* but returns control to *PH*, which deals with them, as will be detailed later.

Example 3 *Consider the transaction* $[a, b, c, d, e]$ *that records a salesperson's actions* [(a) new service promotion, (b) current service cost inquiry, (c) cheaper service promotion, (d) client accepts terms, (e) offer discount], *when marketing a new service. The pattern* $[a, d, e]$ *is sensitive and the pattern* $[d, a]$ *is forbidden, as terms cannot be accepted before new service promotion.* PDPG *produces the permutation* $[a, e, d]$*, which contradicts* $[d, a]$ *and cannot be ruled out, as a discount is typically offered before term acceptance. This is in contrast to permutations following* $[d, a]$*, which are not generated.*

Forbidden patterns can be specified by data owners, based on domain expertise. When data owners have limited or no domain expertise, such patterns can be discovered based on *beliefs* [18] or mined from the original dataset [19]. Unless otherwise stated, we will henceforth assume that all generated permutations contradict the specified forbidden patterns.

IV. Permutation-based Hiding (PH)

This section presents the *Permutation-based Hiding* (*PH*) algorithm. *PH* works in three phases, in which: (I) candidate patterns for lost and ghost are identified, (II) transactions that will be sanitized are selected, and (III) the selected transactions are sanitized. The pseudocode of *PH* describes these phases.

In the first phase (steps 3-7), *PH* iterates over each sensitive pattern and creates the sets of candidates for lost and ghost, as well as a set containing all candidates, for the sensitive pattern (steps 4-6). A set containing the candidates for all sensitive patterns, is also created (step 7). In the next phase (step 8), *PH* uses a function *PST* (to be described later) in order to select transactions that will be sanitized. During the last phase (steps 9-16), *PH* sanitizes each selected transaction. In steps 10-11, the algorithm creates two sets u^- and u^+, which contain the candidates for lost with support $minSup$, and the candidates for ghost with support $minSup - 1$, respectively. The patterns in u^- will become lost, if their support decreases after permutation, whereas those in u^+ will become ghost, if their support increases. In step 13, *PH* sanitizes the sensitive patterns in the selected transaction by applying: (i) *PDPG*, using the candidates in u^- and u^+, (ii) *RR-PDPG*, if step (i) fails and no patterns are forbidden, and (iii) element deletion, if steps (i) and (ii) fail. Next, *PH* updates the transaction and its corresponding sets of candidate patterns, to reflect the hiding (step 14). After that, the algorithm updates the support of patterns in $\mathcal{D}^-(s)$ and $\mathcal{D}^+(s)$ to reflect the sanitization of the transaction, and moves the transaction to the sanitized dataset, which is subsequently returned (steps 15-17).

Algorithm: Permutation-based Hiding (*PH*)

1 $\mathcal{D}_\mathbf{S} \leftarrow \varnothing$
2 $\mathbf{S}' \leftarrow \varnothing$
`// identification of candidate patterns`
3 **foreach** *sensitive pattern* $s \in \mathcal{S}$ **do**
4 $\quad\mathcal{D}^-(s) \leftarrow$ each pattern satisfying condition C1, for every $t \in \mathbf{S}$
5 $\quad\mathcal{D}^+(s) \leftarrow$ each pattern satisfying condition C2, for every $t \in \mathbf{S}$
6 $\quad\mathcal{D}(s) \leftarrow \mathcal{D}^-(s) \cup \mathcal{D}^+(s)$
7 $\quad\mathcal{D}_\mathcal{S} \leftarrow \mathcal{D}_\mathcal{S} \cup \mathcal{D}(s)$
8 $\mathbf{S} \leftarrow PST(\mathbf{S}, \mathbf{S}', minSup, \mathcal{S}, \mathcal{D}_\mathcal{S})$ `// transaction selection`
`// transaction sanitization`
9 **foreach** *transaction* $t \in \mathbf{S}$ **do**
10 $\quad u^- \leftarrow$ each pattern in $\mathcal{D}^{-(t)}(s)$ with support $minSup$ in $\mathbf{S}$, for every sensitive pattern $s \in t$
11 $\quad u^+ \leftarrow$ each pattern in $\mathcal{D}^{+(t)}(s)$ with support $minSup - 1$ in $\mathbf{S}$, for every sensitive pattern $s \in t$
12 $\quad$**foreach** *sensitive pattern* $s \in t$ **do**
13 $\quad\quad$Hide s using the following steps ordered w.r.t. their precedence:
i Apply *PDPG* and replace s with the generated permutation.
ii Apply *RR-PDPG* and replace s with the generated permutation, if step i fails and no patterns are forbidden.
iii Apply symbol deletion, if steps i and ii fail.
14 $\quad\quad$Update t, u^-, u^+, $\mathcal{D}^{-(t)}(s)$, and $\mathcal{D}^{+(t)}(s)$ to reflect the hiding.
15 $\quad$Update the support of patterns in $\mathcal{D}^-(s)$ and $\mathcal{D}^+(s)$ to reflect the sanitization of t.
16 $\quad$Move the sanitized transaction t from $\mathbf{S}$ to $\mathbf{S}'$.
17 **return** $\mathbf{S}'$

Algorithm: Potential Side-effects based Transfer (*PST*)

1 $rank(t) \leftarrow 0$, for every transaction $t \in \mathbf{S}$
2 $\mathbf{S}' \leftarrow$ each $t \in \mathbf{S}$ that does not support any sensitive pattern
3 **foreach** *sensitive pattern* $s \in \mathcal{S}$ **do**
4 $\quad\mathcal{D}(s) \leftarrow$ retrieve $\mathcal{D}(s)$ from $\mathcal{D}_\mathcal{S}$
5 $\quad$**foreach** *transaction* $t \in \mathbf{S}$ **do**
6 $\quad\quad rank(t) \leftarrow rank(t) + (|\mathcal{D}(s)| + 1)$
7 Sort transactions in $\mathbf{S}$ in decreasing order of $rank$.
8 **foreach** *transaction* $t \in \mathbf{S}$ **do**
9 $\quad$**if** $sup_{\mathbf{S}'}(s) < minSup - 1$*, for every sensitive pattern* $s \in t$ **then**
10 $\quad\quad\mathbf{S}' \leftarrow \mathbf{S}' \cup t$
11 $\quad\quad\mathbf{S} \leftarrow \mathbf{S} \setminus t$
12 **return** $\mathbf{S}$

PH sanitizes a sensitive pattern using symbol deletion only in the extreme case, in which there is not a sufficient number of available permutations to apply. Thus, deletion is a rare operation, employed only for very short patterns (e.g., patterns of size two that have only one available permutation). *PH* aims at minimizing the distortion caused by this operation, by deleting the symbol with the largest multiplicity in the transaction, or the largest support in the dataset, if all symbols in the pattern have multiplicity one.

We now describe the *Potential Side-effect based Transfer* (*PST*) function. *PST* transfers two types of transactions, which will not be sanitized, to the dataset $\mathbf{S}$: (I) those that do not support any sensitive pattern, and (II) those whose transfer does not create frequent sensitive patterns in $\mathbf{S}'$. Also, to preserve data utility, *PST* favors transactions of type II that could lead to many side-effects. After transferring transactions of type I, *PST* iterates over sensitive patterns, and retrieves the set of candidates for lost and ghost, for a sensitive pattern (steps 2-4). The size of this set is an upper bound on the

number of side-effects, which would be caused by permuting the pattern. Thus, *PST* ranks all transactions in $\mathbf{S}$, so that transactions that support many patterns with large bound, obtain a higher rank (steps 5-6). Then, *PST* sorts transactions in decreasing order of rank, and transfers each transaction t of type II, if every sensitive pattern supported by t has a support of less than $minSup - 1$ (steps 7-11). Last, *PST* returns $\mathbf{S}$.

The worst case time complexity of *PH* is dominated by finding candidates for lost and ghost, among the obtained frequent patterns. Thus, *PH* needs $O(|\mathbf{S}| \cdot |\mathcal{F}_{\mathbf{S},sup}| \cdot |f|)$ time, where $|\mathbf{S}|$ is the size of the original dataset, $|\mathcal{F}_{\mathbf{S},sup}|$ is the number of obtained frequent patterns, and $|f|$ is the average length of the obtained frequent patterns.

V. Related work

Our work follows the *knowledge hiding* direction of privacy-preserving data mining, whose goal is to prevent the discovery of sensitive patterns through mining. Most research in this area focuses on hiding itemsets, and association or classification rules [4], [15]. Unlike these works, we consider sequential patterns that are more challenging to hide than itemsets, due to their complex semantics.

Abul et al. [1] introduced the problem of sequential pattern hiding and proposed a deletion-based method that minimally affects the support of nonsensitive patterns. Their method does not focus on preventing side-effects and may fail to construct high-quality solutions [8]. Gkoulalas-Divanis et al. [8] developed Side-effect Based Sequence Hiding (*SBSH*), an algorithm that addresses these limitations. *SBSH* identifies symbols to delete, by performing search on graphs that represent matchings between sensitive patterns and transactions. However, due to the high computational cost of graph search, *SBSH* searches only a small part of the graph and does not minimize the distortion needed to avoid side-effects. Besides employing permutation, *PH* addresses the shortcomings of *SBSH* to better preserve data utility. Specifically, it avoids the expensive graph search, while incurring minimal side-effects and distortion by: (I) favoring the sanitization of transactions that incur a small number of side-effects, through the use of *PST*, and (II) replacing sensitive patterns with permutations that prevent side-effects and minimize Cayley distance, through the use of *PDPG* and *RR-PDPG*.

An orthogonal direction of privacy-preserving data mining is *anonymization*, which aims to prevent the disclosure of individuals' private or sensitive information. Anonymization approaches for sequential data have been considered in [5], [14]. These approaches cannot be applied to solve our problem due to their different privacy and utility objectives.

Dataset	$\|\mathbf{S}\|$	$\|\mathcal{A}\|$	Avg. $\|s^{(i)}\|$	$minSup$
MSN	989818	17	5.7	3711,**4949**,9898,19796
BMS	59602	497	2.5	**59**
TRUCKS	273	100	20.1	**20**,40,50,60,67,80

TABLE I: Characteristics of datasets and $minSup$ values

VI. Experiments

We compare *PH* against *SBSH* [8], the state-of-the-art method in terms of data utility and efficiency. We capture data utility by considering sequence mining, frequent itemset mining, as well the difference between the distribution of the support of items (respectively, frequent itemsets) in the original and the sanitized data. Sequential patterns were mined using PrefixSpan [16], which was applied with different $minSup$ and to a varying number of sequences of different lengths. We use the notation $\binom{n}{l}$ to refer to the hiding of n sensitive patterns of length l. Three benchmark datasets, MSNBC (*MSNBC*) from http://fimi.cs.helsinki.fi, BMS-WebView-1 (*BMS*) [21], and TRUCKS (*TRU*) from http://www.chorochronos.org, have been used, as in [1], [8]. Table I shows the characteristics of these datasets and the $minSup$ thresholds used to mine and sanitize them (default values appear in bold). All experiments ran on an Intel Xeon at 2.4 GHz with 12GB of RAM.

Data utility We first demonstrate that *PH* permits more accurate sequential pattern mining than *SBSH*. Fig. 5 shows the percentage of side-effects incurred when sets of patterns of varying length are hidden. As can be seen, the side-effects for *PH* are fewer by 21%, 24% and 28% on average, in *BMS*, *TRU*, and *MSNBC*, respectively. Furthermore, *PH* incurred very few or no ghosts (less than 1.5% in *TRU* and *MSNBC*, when many long patterns were hidden, and 0% in *BMS*). This verifies the effectiveness of *PDPG*.

Next, we demonstrate that *PH* produces significantly more useful data than *SBSH* for tasks beyond sequential pattern mining. In Fig. 6, it can be seen that *PH* preserved at least 77% more frequent itemsets than *SBSH* (on average). On the other hand, up to 20%, 33%, and 13% of frequent itemsets are lost when *SBSH* is applied to *BMS*, *TRU*, and *MSNBC*. Interestingly, the percentage of frequent itemsets lost by *PH* is low, even when there are many sensitive patterns of length 3, which have a small number of available permutations, and it becomes negligible for longer patterns.

In addition, *PH* allows more accurate estimation of the support of items and frequent itemsets, which is important in various tasks. This can be seen in Fig. 7, which shows the KL-divergence scores for *PH* and *SBSH*. KL-divergence measures support estimation accuracy, based on the difference between the distribution of the support of items (or frequent itemsets) in the original and sanitized data [12]. Note that *PH* retained the support distribution of items and frequent items in *BMS*, and achieved at least 10.5 and up 42 times lower KL-divergence scores than *SBSH*. Similar results were obtained for frequent itemsets, mined at the same support threshold as in the previous experiments, and are shown in Fig. 8. The good performance of *PH* in the experiments of Figs. 6 and 7 highlights the benefit of permutation. Observe in Fig. 9(a) that *SBSH* deleted a small percentage of items to sanitize *BMS*, but this had a negative impact on utility. On the contrary, *PH* deleted no items.

Then, we measured utility as a function of $minSup$ (the same threshold was used in pattern mining and hiding). Fig. 9(b) shows the KL-divergence scores for items and frequent items, and Fig. 9(c) for frequent itemsets, when $minSup$ varies. Note that, unlike *SBSH*, *PH* preserved the support of items and frequent items, as well as all frequent itemsets. Similar results were observed for *BMS* and *MSNBC* (omitted).

Last, we measured the impact of forbidden patterns on data utility. Fig. 9(d) shows the side-effects incurred, when 50 sensitive patterns of length 5 were hidden, and a varying

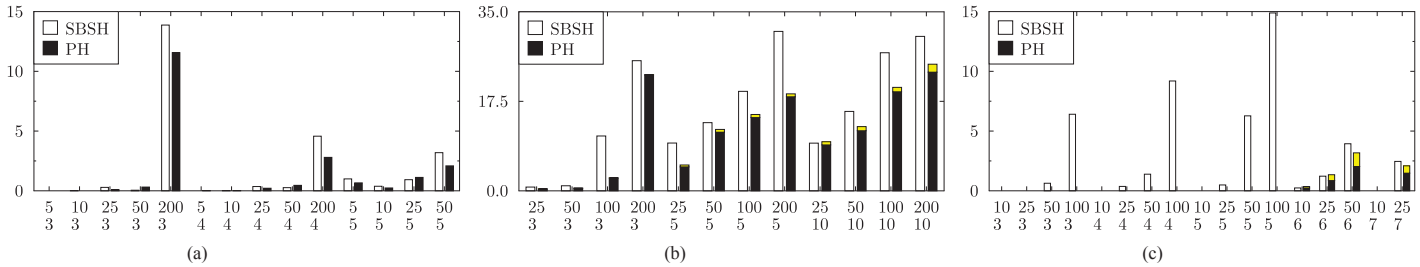


Fig. 5: *Side-effects* (%) for (a)*BMS*, (b)*TRU*, and (c)*MSNBC* (ghosts are shown in yellow).

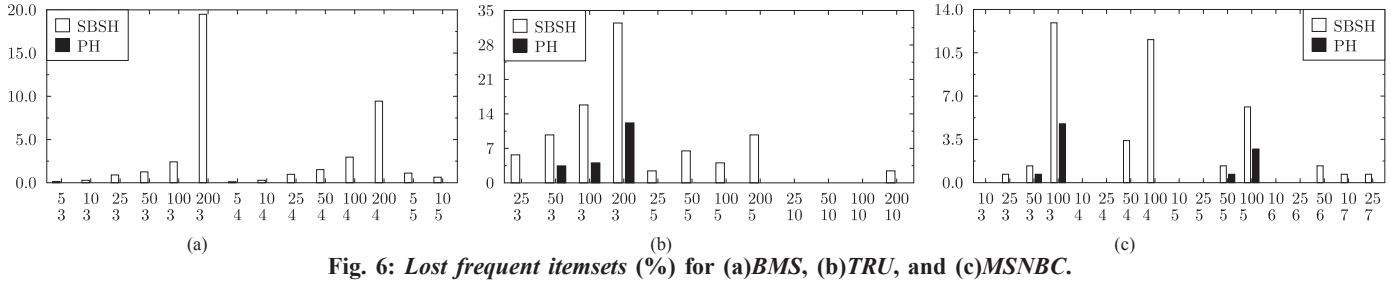


Fig. 6: *Lost frequent itemsets* (%) for (a)*BMS*, (b)*TRU*, and (c)*MSNBC*.

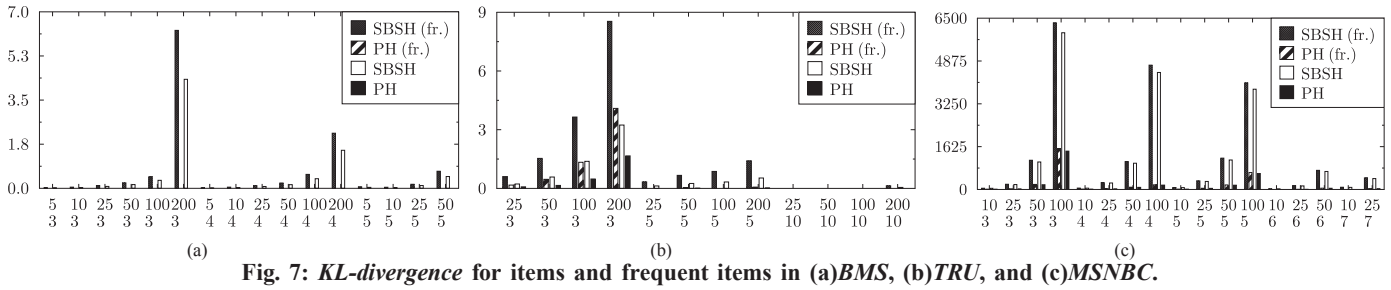


Fig. 7: *KL-divergence* for items and frequent items in (a)*BMS*, (b)*TRU*, and (c)*MSNBC*.

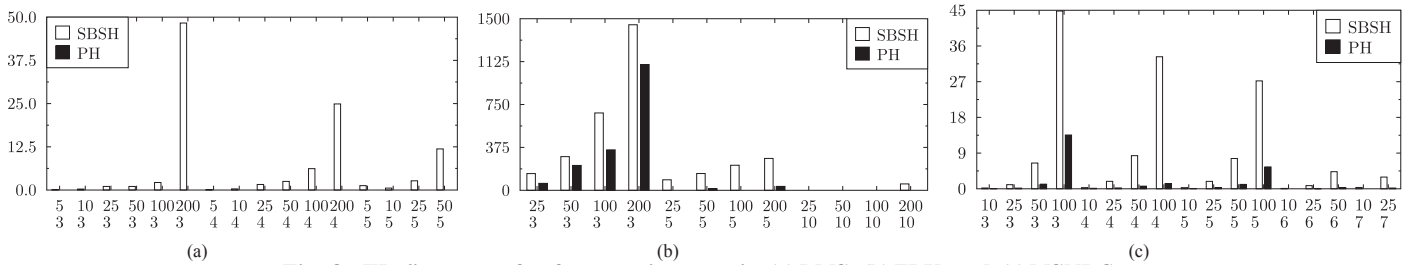


Fig. 8: *KL-divergence* for frequent itemsets in (a)*BMS*, (b)*TRU*, and (c)*MSNBC*.

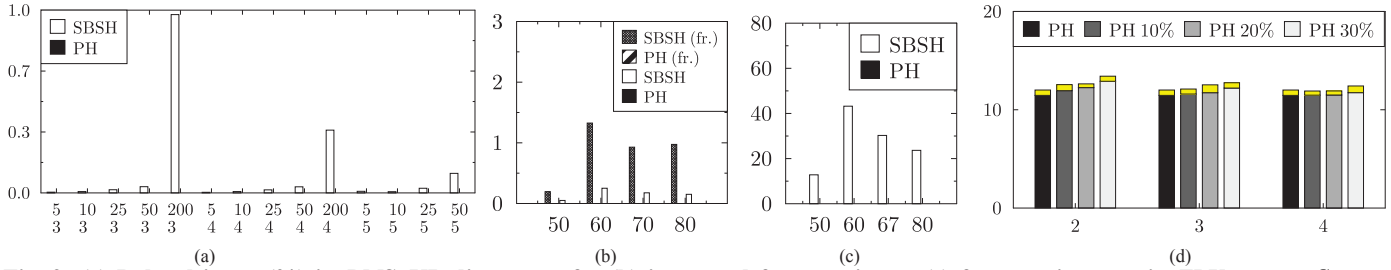


Fig. 9: (a) Deleted items (%) in *BMS*. KL-divergence for (b) items and frequent items, (c) frequent itemsets in *TRU* vs. $minSup$. (d) *Side-effects* (%) for *TRU* when 10%, 20%, and 30% of patterns with length 2, 3, or 4 are forbidden (ghosts are shown in yellow).

percentage of their randomly selected permutations with length in $[2, 4]$ are forbidden. *PH* incurred a small number of side-effects, even in demanding scenarios, in which many short patterns are forbidden (i.e., few permutations are possible). Furthermore, as shown in Fig. 10(a), the percentage of lost frequent itemsets for *PH* is lower than that for *SBSH*.

Efficiency Experiments with runtime are reported in Figs. 10(b), 10(c), and 10(d). *PH* was slower than *SBSH*, due to the significantly larger problem space it considers. However, *PH* scaled linearly with the number of sensitive patterns and took less than 10 seconds. The results for runtime as a function of $minSup$, are shown in Fig. 10(d). Both algorithms needed

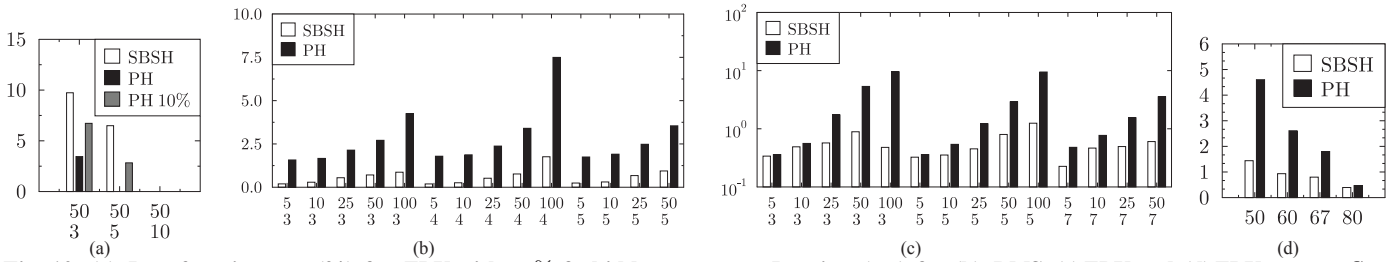


Fig. 10: (a) *Lost freq. itemsets* (%) for *TRU* with 10% forbidden patterns. *Runtime* (sec) for (b) *BMS*, (c)*TRU* and (d)*TRU* vs. $minSup$.

more time when $minSup$ was smaller, because there are more nonsensitive patterns that need to be dealt with, and *SBSH* was 2.3 times faster on average. Moreover, the overhead of *PH* for dealing with forbidden patterns was minimal (e.g., in the order of milliseconds in the experiments in Figs. 9(d) and 10(a)).

VII. Preventing background knowledge attacks

Our work follows the most common, *minimum harm* approach [9], [10], which dictates hiding sensitive patterns below a data owner specified $minSup$ threshold to help data utility. This approach makes the reasonable assumption that sensitive patterns are not exposed when their support is below $minSup$, as they model *unexpected* knowledge that is not discoverable from external datasets, commonly considered in *microdata* anonymization [12]. Nevertheless, it is worth noting that our permutation-based framework can offer protection against attackers who may (I) discover a sensitive pattern s when its support is lower, but "close" to $minSup$, and there are "few" nonsensitive patterns with the same support as s, and/or (II) exclude certain permutations of s from consideration.

In fact, due to the factorial growth of the number of permutations of a sensitive pattern s with its length, protection against both classes of attackers can be provided by generating a "sufficient" number of permutations by *PDPG* and using them to replace s. More concretely, a *maximum uncertainty* approach that hides s using the maximum possible number of different permutations (i.e., $min(sup(s) - 1, |s|! - 1)$) can be adopted. This approach offers stronger privacy but lower utility than the minimum harm approach, because it aims at minimizing the probability of discovering s among nonsensitive, infrequent patterns (the permutations of s are among these patterns), as well as the probability of inferring s by excluding some of its permutations. Alternatively, a more flexible, *bounded uncertainty* approach can be employed to hide s by replacing it with at least c different permutations, all having support at most ℓ, where $c \geq 1$ and $\ell < minSup$ are data owner specified parameters. This approach takes a middle line between the maximum uncertainty and minimum harm approach, which are mapped to it by setting c to its maximum value and 1, respectively. Both approaches can be easily implemented by imposing restrictions to *PDPG*.

VIII. Conclusions

Employing deletion to hide sensitive sequential patterns reduces the usefulness of data in sequential pattern mining and tasks based on itemset properties. In response, we proposed a novel, permutation-based approach and the *PH* algorithm for sanitizing sequence data with minimal side-effects and distortion. Our experiments verified that *PH* permits more effective data analysis than the state-of-the-art method.

IX. Acknowledgments

Robert Gwadera was supported by the EU project OpenIoT (ICT 287305). Grigorios Loukides was supported by a Research Fellowship from the Royal Academy of Engineering.

References

[1] O. Abul, F. Bonchi, and F. Giannotti. Hiding sequential and spatiotemporal patterns. *TKDE*, 22(12):1709–1723, 2010.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, 1995.

[4] K. Chen and L. Liu. Privacy preserving data classification with rotation perturbation. In *ICDM*, pages 589–592, 2005.

[5] R. Chen, B. Fung, B. Desai, and N. Sossou. Differentially private transit data publication: a case study on the Montreal transportation system. In *KDD*, pages 213–221, 2012.

[6] G. Das and N. Zhang. Privacy risks in health databases from aggregate disclosure. In *PETRA*, pages 1–4, 2009.

[7] B. C. Fung, K. Wang, and M. Ester. Hierarchical document clustering using frequent itemsets. In *SDM*, pages 59–70, 2003.

[8] A. Gkoulalas-Divanis and G. Loukides. Revisiting sequential pattern hiding to enhance utility. In *KDD*, pages 1316–1324, 2011.

[9] A. Gkoulalas-Divanis and V. S. Verykios. Exact knowledge hiding through database extension. *TKDE*, 21(5):699–713, 2009.

[10] A. Gkoulalas-Divanis and V. S. Verykios. *Association Rule Hiding for Data Mining*. Springer, 2010.

[11] R. Gwadera, G. Antonini, and A. Labbi. Mining actionable partial orders in collections of sequences. In *ECML/PKDD*, 2011.

[12] D. Kifer and J. Gehrke. Injecting utility into anonymized datasets. In *SIGMOD*, pages 217–228, 2006.

[13] J. Marden. *Analyzing and Modeling Rank Data*. Chapman&Hall, 1995.

[14] N. Mohammed, B. Fung, and M. Debbabi. Walking in the crowd: anonymizing trajectory data for pattern analysis. In *CIKM*, 2009.

[15] S. R. M. Oliveira and O. R. Zaïane. Protecting sensitive knowledge by data sanitization. In *ICDM*, pages 211–218, 2003.

[16] J. Pei, J.Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *ICDE*, pages 215–224, 2001.

[17] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison Wesley, 1996.

[18] M. Spiliopoulou. Managing interesting rules in sequence mining. In *PKDD*, pages 554–560, 1999.

[19] P. Sun, S. Chawla, and B. Arunasalam. Mining for outliers in sequential databases. In *SDM*, pages 94–105, 2006.

[20] H. S. Wilf. East side, west side . . . - an introduction to combinatorial families-with maple programming, 1999.

[21] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *KDD*, pages 401–406, 2001.