String Sanitization: A Combinatorial Approach

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

[1]University of Milano-Bicocca
[2]King's College London
[3]University of Pisa
[4]ERABLE Team, INRIA, Lyon
[5]CWI, Amsterdam

CWI-INRIA Workshop 2019

Definitions and Motivation

Definitions and Motivation

An **alphabet** Σ is a finite set whose elements are called letters.

Definitions and Motivation

An **alphabet** Σ is a finite set whose elements are called letters.

A **string** W is a sequence of letters over Σ.

Definitions and Motivation

An **alphabet** Σ is a finite set whose elements are called letters.
A **string** W is a sequence of letters over Σ.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbaab}$$

Definitions and Motivation

An **alphabet** Σ is a finite set whose elements are called letters.
A **string** W is a sequence of letters over Σ.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbaab}$$

- W may represent location history, search queries, DNA sequence, etc.

Definitions and Motivation

An **alphabet** Σ is a finite set whose elements are called letters.
A **string** W is a sequence of letters over Σ.

$$\Sigma = \{\mathrm{a}, \mathrm{b}\} \qquad W = \texttt{aabaaaababbbaab}$$

- W may represent location history, search queries, DNA sequence, etc.
- W fuels up location-based, web analytics, or bioinformatics apps.

Definitions and Motivation

An **alphabet** Σ is a finite set whose elements are called letters.
A **string** W is a sequence of letters over Σ.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbaab}$$

- W may represent location history, search queries, DNA sequence, etc.
- W fuels up location-based, web analytics, or bioinformatics apps.
- Dissemination may expose patterns modeling confidential knowledge.

Definitions and Motivation

An **alphabet** Σ is a finite set whose elements are called letters.
A **string** W is a sequence of letters over Σ.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbaab}$$

- W may represent location history, search queries, DNA sequence, etc.
- W fuels up location-based, web analytics, or bioinformatics apps.
- Dissemination may expose patterns modeling confidential knowledge.
- We call these patterns sensitive.

Definitions and Motivation

An **alphabet** Σ is a finite set whose elements are called letters.
A **string** W is a sequence of letters over Σ.

$$\Sigma = \{\mathtt{a}, \mathtt{b}\} \qquad W = \mathtt{aabaaaababbbaab}$$

- W may represent location history, search queries, DNA sequence, etc.
- W fuels up location-based, web analytics, or bioinformatics apps.
- Dissemination may expose patterns modeling confidential knowledge.
- We call these patterns sensitive.

W = aabaaaababbbaab

Definitions and Motivation

An **alphabet** Σ is a finite set whose elements are called letters.
A **string** W is a sequence of letters over Σ.

$$\Sigma = \{a, b\} \qquad W = aabaaaababbbaab$$

- W may represent location history, search queries, DNA sequence, etc.
- W fuels up location-based, web analytics, or bioinformatics apps.
- Dissemination may expose patterns modeling confidential knowledge.
- We call these patterns sensitive.

W = aabaaaababbbaab

The goal: String sanitization

Conceal **sensitive patterns** in W while maintaining **data utility**.

The Model: Combinatorial String Dissemination

The Model: Combinatorial String Dissemination

Combinatorial String Dissemination (CSD) model

Given **constraints** and **properties**, determine the **edit operations** to be applied to W so that the properties are satisfied subject to the constraints.

The Model: Combinatorial String Dissemination

Combinatorial String Dissemination (CSD) model

Given **constraints** and **properties**, determine the **edit operations** to be applied to W so that the properties are satisfied subject to the constraints.

Our CSD setting

The Model: Combinatorial String Dissemination

Combinatorial String Dissemination (CSD) model

Given **constraints** and **properties**, determine the **edit operations** to be applied to W so that the properties are satisfied subject to the constraints.

Our CSD setting

- Given W and a set of **length-**k sensitive patterns **construct** X:

The Model: Combinatorial String Dissemination

Combinatorial String Dissemination (CSD) model

Given **constraints** and **properties**, determine the **edit operations** to be applied to W so that the properties are satisfied subject to the constraints.

Our CSD setting

- Given W and a set of **length-**k sensitive patterns **construct** X:
  - **C1** No length-k sensitive pattern occurs in X.

The Model: Combinatorial String Dissemination

Combinatorial String Dissemination (CSD) model

Given **constraints** and **properties**, determine the **edit operations** to be applied to W so that the properties are satisfied subject to the constraints.

Our CSD setting

- Given W and a set of **length-**k sensitive patterns **construct** X:
  - **C1** No length-k sensitive pattern occurs in X.
  - **P1** The order of length-k non-sensitive patterns is preserved in X.

The Model: Combinatorial String Dissemination

Combinatorial String Dissemination (CSD) model

Given **constraints** and **properties**, determine the **edit operations** to be applied to W so that the properties are satisfied subject to the constraints.

Our CSD setting

- Given W and a set of **length-**k sensitive patterns **construct** X:
  - **C1** No length-k sensitive pattern occurs in X.
  - **P1** The order of length-k non-sensitive patterns is preserved in X.

    $\Rightarrow$ No utility loss for tasks based on sequentiality.

The Model: Combinatorial String Dissemination

Combinatorial String Dissemination (CSD) model

Given **constraints** and **properties**, determine the **edit operations** to be applied to W so that the properties are satisfied subject to the constraints.

Our CSD setting

- Given W and a set of **length-**k sensitive patterns **construct** X:
  - **C1** No length-k sensitive pattern occurs in X.
  - **P1** The order of length-k non-sensitive patterns is preserved in X.
    $\Rightarrow$ No utility loss for tasks based on sequentiality.
  - **P2** The frequency of length-k non-sensitive patterns is preserved in X.

The Model: Combinatorial String Dissemination

Combinatorial String Dissemination (CSD) model

Given **constraints** and **properties**, determine the **edit operations** to be applied to W so that the properties are satisfied subject to the constraints.

Our CSD setting

- Given W and a set of **length-**k sensitive patterns **construct** X:
  - **C1** No length-k sensitive pattern occurs in X.
  - **P1** The order of length-k non-sensitive patterns is preserved in X.
    $\Rightarrow$ No utility loss for tasks based on sequentiality.
  - **P2** The frequency of length-k non-sensitive patterns is preserved in X.
    $\Rightarrow$ No utility loss for tasks based on frequency.

Our Results

Our Results

TFS (Total order, Frequency, Sanitization) problem

Construct the **shortest** string X that satisfies **P1**, **P2**, and **C1**.

Our Results

TFS (Total order, Frequency, Sanitization) problem

Construct the **shortest** string X that satisfies **P1**, **P2**, and **C1**.

Example. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $W = \mathtt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\{\mathtt{baaa,aaaa,bbaa}\}$.

Our Results

TFS (Total order, Frequency, Sanitization) problem

Construct the **shortest** string X that satisfies **P1**, **P2**, and **C1**.

Example. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $W = \mathtt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\{\mathtt{baaa}, \mathtt{aaaa}, \mathtt{bbaa}\}$.

W = aabaaaababbbaab

Our Results

TFS (Total order, Frequency, Sanitization) problem

Construct the **shortest** string X that satisfies **P1**, **P2**, and **C1**.

Example. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $W = \mathtt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\{\mathtt{baaa,aaaa,bbaa}\}$.

W = aabaaaababbbaab

$$X = \mathtt{aabaa\#aaababbba\#baab}$$

Our Results

TFS (Total order, Frequency, Sanitization) problem

Construct the **shortest** string X that satisfies **P1**, **P2**, and **C1**.

Example. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $W = \mathtt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\{\mathtt{baaa}, \mathtt{aaaa}, \mathtt{bbaa}\}$.

W = aabaaaababbbaab

$$X = \mathtt{aabaa\#aaababbba\#baab}$$

where # is a letter not in Σ. □

Our Results

TFS (Total order, Frequency, Sanitization) problem

Construct the **shortest** string X that satisfies **P1**, **P2**, and **C1**.

Example. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $W = \mathtt{aabaaaababbbaab}$, $k = 4$, and the set of sensitive patterns be $\{\mathtt{baaa}, \mathtt{aaaa}, \mathtt{bbaa}\}$.

W = aabaaaababbbaab

$$X = \mathsf{aabaa\#aaababbba\#baab}$$

where # is a letter not in Σ. $\square$

Theorem

The **length** *of* X *is in* $\Theta(k|W|)$. **TFS-ALGO** *solves* **TFS** *in the optimal* $O(k|W|)$ *time.*

Our Results

Our Results

Could we generally hope for a shorter string?

Our Results

Could we generally hope for a shorter string? Relax the total order (**P1**).

Our Results

Could we generally hope for a shorter string? Relax the total order (**P1**).
Employ Π**1**: order of length-k patterns in-between #s remains unchanged.

Our Results

Could we generally hope for a shorter string? Relax the total order (**P1**).
Employ Π**1**: order of length-k patterns in-between #s remains unchanged.

PFS (Partial order, Frequency, Sanitization) problem

Construct a **shortest** string Y that satisfies Π**1**, **P2**, and **C1**.

Our Results

Could we generally hope for a shorter string? Relax the total order (**P1**).
Employ Π**1**: order of length-k patterns in-between #s remains unchanged.

PFS (Partial order, Frequency, Sanitization) problem

Construct a **shortest** string Y that satisfies Π**1**, **P2**, and **C1**.

W = aabaaaababbbaab

Our Results

Could we generally hope for a shorter string? Relax the total order (**P1**). Employ **Π1**: order of length-k patterns in-between #s remains unchanged.

PFS (Partial order, Frequency, Sanitization) problem

Construct a **shortest** string Y that satisfies **Π1**, **P2**, and **C1**.

W = aabaaaababbbaab

$X =$ aabaa#aaababbba#baab

Our Results

Could we generally hope for a shorter string? Relax the total order (**P1**).
Employ Π**1**: order of length-k patterns in-between #s remains unchanged.

PFS (Partial order, Frequency, Sanitization) problem

Construct a **shortest** string Y that satisfies Π**1**, **P2**, and **C1**.

W = aabaaaababbbaab

$$X = \text{aabaa\#aaababbba\#baab}$$

$$Y = \text{aaababbba\#aabaab}$$

Our Results

Could we generally hope for a shorter string? Relax the total order (**P1**). Employ **Π1**: order of length-k patterns in-between #s remains unchanged.

PFS (Partial order, Frequency, Sanitization) problem

Construct a **shortest** string Y that satisfies **Π1**, **P2**, and **C1**.

W = aabaaaababbbaab

$$X = \text{aabaa\#aaababbba\#baab}$$
$$Y = \text{aaababbba\#aabaab}$$

Theorem

PFS-ALGO *solves* **PFS** *in the optimal* $O(|W| + |Y|)$ *time.*

Our Results

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

MCSR (Minimum-Cost Separators Replacement)

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

MCSR (Minimum-Cost Separators Replacement)

Construct a string Z by deleting or replacing each # in Y:

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

MCSR (Minimum-Cost Separators Replacement)

Construct a string Z by deleting or replacing each # in Y:
(I) the total **weight** of letter **replacements** is bounded by θ;

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

MCSR (Minimum-Cost Separators Replacement)

Construct a string Z by deleting or replacing each # in Y:
(I) the total **weight** of letter **replacements** is bounded by θ;
(II) the total **cost** of τ-**ghost** occurrences of in Z is **minimum**;

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

MCSR (Minimum-Cost Separators Replacement)

Construct a string Z by deleting or replacing each # in Y:
(I) the total **weight** of letter **replacements** is bounded by θ;
(II) the total **cost** of τ-**ghost** occurrences of in Z is **minimum**;
(III) **C1** is satisfied.

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

MCSR (Minimum-Cost Separators Replacement)

Construct a string Z by deleting or replacing each # in Y:
(I) the total **weight** of letter **replacements** is bounded by θ;
(II) the total **cost** of τ-**ghost** occurrences of in Z is **minimum**;
(III) **C1** is satisfied.

$$Y = \mathtt{aaababbba\#aabaab}$$

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

MCSR (Minimum-Cost Separators Replacement)

Construct a string Z by deleting or replacing each # in Y:
(I) the total **weight** of letter **replacements** is bounded by θ;
(II) the total **cost** of τ-**ghost** occurrences of in Z is **minimum**;
(III) **C1** is satisfied.

$$Y = \texttt{aaababbba\#aabaab}$$
$$Z = \texttt{aaababbba}\mathbf{b}\texttt{aabaab}$$

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

MCSR (Minimum-Cost Separators Replacement)

Construct a string Z by deleting or replacing each # in Y:
(I) the total **weight** of letter **replacements** is bounded by θ;
(II) the total **cost** of τ-**ghost** occurrences of in Z is **minimum**;
(III) **C1** is satisfied.

$$Y = \texttt{aaababbba\#aabaab}$$
$$Z = \texttt{aaababbba}\mathbf{b}\texttt{aabaab}$$

Theorem

MCSR *is NP-hard via the Multiple-Choice Knapsack (MCK).*

Our Results

Observation: #s in Y may reveal the location of sensitive patterns.
Replacing #s with alphabet letters creates spurious patterns:
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ after sanitization.

MCSR (Minimum-Cost Separators Replacement)

Construct a string Z by deleting or replacing each # in Y:
(I) the total **weight** of letter **replacements** is bounded by θ;
(II) the total **cost** of τ-**ghost** occurrences of in Z is **minimum**;
(III) **C1** is satisfied.

$$Y = \texttt{aaababbba\#aabaab}$$
$$Z = \texttt{aaababbba}\mathbf{b}\texttt{aabaab}$$

Theorem

MCSR *is NP-hard via the Multiple-Choice Knapsack (MCK).*

We also develop **MCSR-ALGO**, an effective heuristic to solve **MCSR**.

String Sanitization: TFS-ALGO

String Sanitization: TFS-ALGO

- Read W from left to right.

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: `...baaa` → `...baa#aaa`

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: $\ldots$`baaa` $\rightarrow$ $\ldots$`baa#aaa` (because we must hide `baaa`).

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: $\ldots$baaa $\rightarrow$ $\ldots$baa#aaa (because we must hide baaa).
  - **R2**: $\ldots\underline{\texttt{aaa}}\texttt{\#}\underline{\texttt{aaa}}\texttt{b} \rightarrow \ldots\texttt{aaab}$

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: $\ldots$`baaa` $\rightarrow$ $\ldots$`baa#aaa` (because we must hide `baaa`).
  - **R2**: $\ldots\underline{\texttt{aaa}}\texttt{\#}\underline{\texttt{aaa}}\texttt{b} \rightarrow \ldots\texttt{aaab}$ (because a shorter string is possible).

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: $\ldots\texttt{baaa} \rightarrow \ldots\texttt{baa\#aaa}$ (because we must hide $\texttt{baaa}$).
  - **R2**: $\ldots\underline{\texttt{aaa}}\texttt{\#}\underline{\texttt{aaa}}\texttt{b} \rightarrow \ldots\texttt{aaab}$ (because a shorter string is possible).

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: $\ldots$`baaa` $\rightarrow$ $\ldots$`baa#aaa` (because we must hide `baaa`).
  - **R2**: $\ldots\underline{\texttt{aaa}}\texttt{\#}\underline{\texttt{aaa}}\texttt{b} \rightarrow \ldots\texttt{aaab}$ (because a shorter string is possible).

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

- Implementing **R1** and **R2** carefully produces X in $O(k|W|)$ time.

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: $\ldots$`baaa` $\rightarrow$ $\ldots$`baa#aaa` (because we must hide `baaa`).
  - **R2**: $\ldots$`aaa#aaab` $\rightarrow$ $\ldots$`aaab` (because a shorter string is possible).

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

- Implementing **R1** and **R2** carefully produces X in $O(k|W|)$ time.
- This is optimal because the length of X is $\Omega(k|W|)$.

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: `...baaa` $\rightarrow$ `...baa#aaa` (because we must hide `baaa`).
  - **R2**: `...aaa#aaab` $\rightarrow$ `...aaab` (because a shorter string is possible).

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

- Implementing **R1** and **R2** carefully produces X in $O(k|W|)$ time.
- This is optimal because the length of X is $\Omega(k|W|)$.
- **Tightness** proof:

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: $\ldots$`baaa` $\rightarrow$ $\ldots$`baa#aaa` (because we must hide `baaa`).
  - **R2**: $\ldots\underline{\texttt{aaa}}\texttt{\#}\underline{\texttt{aaa}}\texttt{b}$ $\rightarrow$ $\ldots$`aaab` (because a shorter string is possible).

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

- Implementing **R1** and **R2** carefully produces X in $O(k|W|)$ time.
- This is optimal because the length of X is $\Omega(k|W|)$.
- **Tightness** proof:
  - Construct the de Bruijn string W of order $k-1$ over Σ.

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: $\ldots$`baaa` $\rightarrow$ $\ldots$`baa#aaa` (because we must hide `baaa`).
  - **R2**: $\ldots\underline{\texttt{aaa}}\texttt{\#}\underline{\texttt{aaa}}\texttt{b}$ $\rightarrow$ $\ldots$`aaab` (because a shorter string is possible).

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

- Implementing **R1** and **R2** carefully produces X in $O(k|W|)$ time.
- This is optimal because the length of X is $\Omega(k|W|)$.
- **Tightness** proof:
  - Construct the de Bruijn string W of order $k-1$ over Σ.
  - Assign every other consecutive length-k substring to be sensitive.

String Sanitization: TFS-ALGO

- Read W from left to right.
- If the length-k substring read is non-sensitive append it to X.
- Otherwise:
  - **R1**: $\ldots\texttt{baaa} \rightarrow \ldots\texttt{baa\#aaa}$ (because we must hide $\texttt{baaa}$).
  - **R2**: $\ldots\underline{\texttt{aaa}}\texttt{\#}\underline{\texttt{aaa}}\texttt{b} \rightarrow \ldots\texttt{aaab}$ (because a shorter string is possible).

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

- Implementing **R1** and **R2** carefully produces X in $O(k|W|)$ time.
- This is optimal because the length of X is $\Omega(k|W|)$.
- **Tightness** proof:
  - Construct the de Bruijn string W of order $k-1$ over Σ.
  - Assign every other consecutive length-k substring to be sensitive.
  - Then X is of length $\Omega(k|W|)$ because no overlap exists.

String Sanitization: PFS-ALGO

String Sanitization: PFS-ALGO

- If blocks in-between #s **overlap** by $k-1$ letters, then we can further apply **R2** while still satisfying **Π1**.

String Sanitization: PFS-ALGO

- If blocks in-between #s **overlap** by $k-1$ letters, then we can further apply **R2** while still satisfying **Π1**.

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

Y = aaababbba#aabaab

String Sanitization: PFS-ALGO

- If blocks in-between #s **overlap** by $k-1$ letters, then we can further apply **R2** while still satisfying Π**1**.

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

Y = aaababbba#aabaab

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete.

String Sanitization: PFS-ALGO

- If blocks in-between #s **overlap** by $k-1$ letters, then we can further apply **R2** while still satisfying **Π1**.

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

Y = aaababbba#aabaab

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete.
- **Observation**: Overlaps allowed are of fixed length $k-1$.

String Sanitization: PFS-ALGO

- If blocks in-between #s **overlap** by $k-1$ letters, then we can further apply **R2** while still satisfying **Π1**.

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

Y = aaababbba#aabaab

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete.
- **Observation**: Overlaps allowed are of fixed length $k-1$.
- **Idea**:

String Sanitization: PFS-ALGO

- If blocks in-between #s **overlap** by $k-1$ letters, then we can further apply **R2** while still satisfying **Π1**.

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

Y = aaababbba#aabaab

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete.
- **Observation**: Overlaps allowed are of fixed length $k-1$.
- **Idea**:
  - Assign **ids** to blocks' prefixes and suffixes of length $k-1$.

String Sanitization: PFS-ALGO

- If blocks in-between #s **overlap** by $k-1$ letters, then we can further apply **R2** while still satisfying **Π1**.

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

Y = aaababbba#aabaab

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete.
- **Observation**: Overlaps allowed are of fixed length $k-1$.
- **Idea**:
  - Assign **ids** to blocks' prefixes and suffixes of length $k-1$.
  - Ignore the middle part of the blocks (it plays no role).

String Sanitization: PFS-ALGO

- If blocks in-between #s **overlap** by $k-1$ letters, then we can further apply **R2** while still satisfying **Π1**.

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

Y = aaababbba#aabaab

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete.
- **Observation**: Overlaps allowed are of fixed length $k-1$.
- **Idea**:
  - Assign **ids** to blocks' prefixes and suffixes of length $k-1$.
  - Ignore the middle part of the blocks (it plays no role).
  - Solve SCS for two-letter strings [Gallant et al., JCSS, 1980].

String Sanitization: PFS-ALGO

- If blocks in-between #s **overlap** by $k-1$ letters, then we can further apply **R2** while still satisfying **Π1**.

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

Y = aaababbba#aabaab

- **Looks bad**: Shortest Common Superstring (SCS) is NP-complete.
- **Observation**: Overlaps allowed are of fixed length $k-1$.
- **Idea**:
  - Assign **ids** to blocks' prefixes and suffixes of length $k-1$.
  - Ignore the middle part of the blocks (it plays no role).
  - Solve SCS for two-letter strings [Gallant et al., JCSS, 1980].
- **Result**: Linear time!

String Sanitization: MSCR-ALGO

String Sanitization: MSCR-ALGO

Create an instance of MCK from an instance of **MCSR**:

String Sanitization: MSCR-ALGO

Create an instance of MCK from an instance of **MCSR**:

- Compute the set of all **candidate ghost** patterns.

String Sanitization: MSCR-ALGO

Create an instance of MCK from an instance of **MCSR**:

- Compute the set of all **candidate ghost** patterns.
- Assign a **cost** to each candidate ghost.

String Sanitization: MSCR-ALGO

Create an instance of MCK from an instance of **MCSR**:

- Compute the set of all **candidate ghost** patterns.
- Assign a **cost** to each candidate ghost.
- Assign a **weight** to each possible letter replacement.

String Sanitization: MSCR-ALGO

Create an instance of MCK from an instance of **MCSR**:

- Compute the set of all **candidate ghost** patterns.
- Assign a **cost** to each candidate ghost.
- Assign a **weight** to each possible letter replacement.

Solve the MCK instance [Pissinger, Eur J Oper Res, 1995].

String Sanitization: MSCR-ALGO

Create an instance of MCK from an instance of **MCSR**:

- Compute the set of all **candidate ghost** patterns.
- Assign a **cost** to each candidate ghost.
- Assign a **weight** to each possible letter replacement.

Solve the MCK instance [Pissinger, Eur J Oper Res, 1995].
Translate the solution back to **MCSR**.

String Sanitization: MSCR-ALGO

Create an instance of MCK from an instance of **MCSR**:

- Compute the set of all **candidate ghost** patterns.
- Assign a **cost** to each candidate ghost.
- Assign a **weight** to each possible letter replacement.

Solve the MCK instance [Pissinger, Eur J Oper Res, 1995].
Translate the solution back to **MCSR**.

W = aabaaaababbbaab

X = aabaa#aaababbba#baab

Y = aaababbba#aabaab

Z = aaababbbabaabaab

Experiments: The Datasets

Experiments: The Datasets

- We evaluate the pipeline **TFS-ALGO**→**PFS-ALGO**→**MCSR-ALGO**, referred to as **TPM**, in terms of *data utility* and *efficiency*.

Experiments: The Datasets

- We evaluate the pipeline **TFS-ALGO**→**PFS-ALGO**→**MCSR-ALGO**, referred to as **TPM**, in terms of *data utility* and *efficiency*.
- Four real datasets (OLD, TRU, MSN, DNA) and a synthetic (SYN).

Experiments: The Datasets

- We evaluate the pipeline **TFS-ALGO**→**PFS-ALGO**→**MCSR-ALGO**, referred to as **TPM**, in terms of *data utility* and *efficiency*.
- Four real datasets (OLD, TRU, MSN, DNA) and a synthetic (SYN).

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

Experiments: The Datasets

- We evaluate the pipeline **TFS-ALGO**→**PFS-ALGO**→**MCSR-ALGO**, referred to as **TPM**, in terms of *data utility* and *efficiency*.
- Four real datasets (OLD, TRU, MSN, DNA) and a synthetic (SYN).

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

- We compared **TPM** against a greedy baseline, referred to as **BA**.

Experiments: The Datasets

- We evaluate the pipeline **TFS-ALGO**$\to$**PFS-ALGO**$\to$**MCSR-ALGO**, referred to as **TPM**, in terms of *data utility* and *efficiency*.
- Four real datasets (OLD, TRU, MSN, DNA) and a synthetic (SYN).

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

- We compared **TPM** against a greedy baseline, referred to as **BA**.
- **BA** replaces #s greedily from left to right based on letter frequencies.

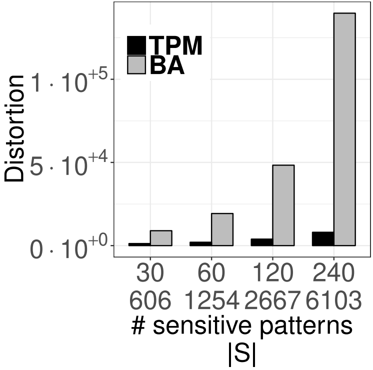
Experiments: Frequency Distortion

Experiments: Frequency Distortion

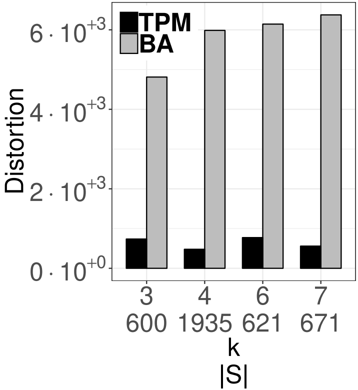
$\sum_U (\mathrm{Freq}_W(U) - \mathrm{Freq}_Z(U))^2$, where U is a non-sensitive pattern.

Experiments: Frequency Distortion

$\sum_U(\text{Freq}_W(U) - \text{Freq}_Z(U))^2$, where U is a non-sensitive pattern.



OLD



OLD

S denotes the set of occurrences of sensitive patterns.

Experiments: Lost and Ghost Patterns

Experiments: Lost and Ghost Patterns

τ-**losts** are patterns with frequency $> \tau$ in W and $\leq \tau$ in Z.

Experiments: Lost and Ghost Patterns

τ-**losts** are patterns with frequency $> \tau$ in W and $\leq \tau$ in Z.

τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ in Z.

Experiments: Lost and Ghost Patterns

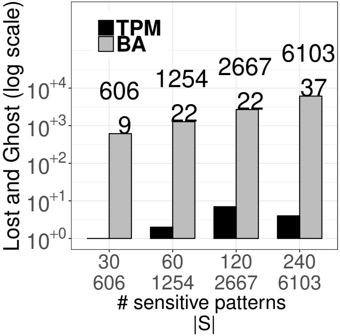
τ-**losts** are patterns with frequency $> \tau$ in W and $\leq \tau$ in Z.

τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ in Z.

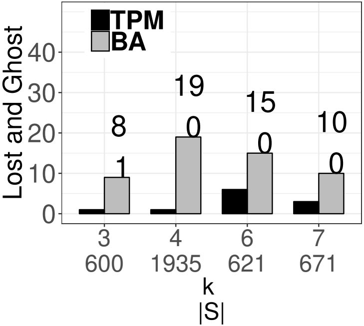
We used $\tau = 20$.

Experiments: Lost and Ghost Patterns

τ-**losts** are patterns with frequency $> \tau$ in W and $\leq \tau$ in Z.
τ-**ghosts** are patterns with frequency $< \tau$ in W and $\geq \tau$ in Z.
We used $\tau = 20$.



OLD



OLD

$^{x}_{y}$ on the top of each bar for **BA** denotes x τ-lost and y τ-ghost.

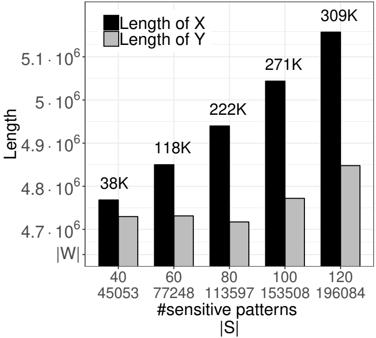
Experiments: Output Size

Experiments: Output Size

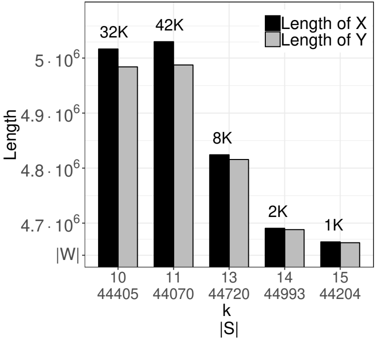
Length of X and Y (output of **TFS-ALGO** and **PFS-ALGO**, resp.).

Experiments: Output Size

Length of X and Y (output of **TFS-ALGO** and **PFS-ALGO**, resp.).



DNA

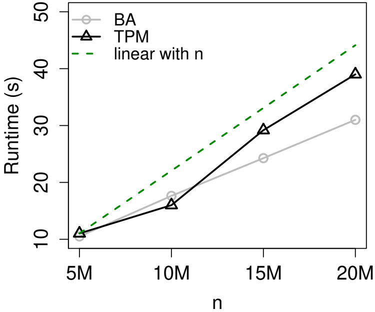


DNA

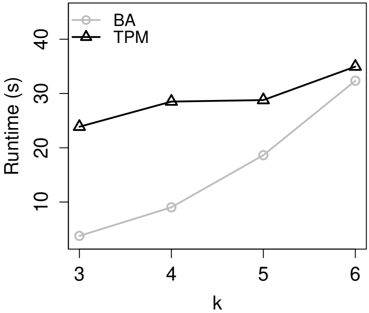
On the top of each pair of bars we plot $|X| - |Y|$.

Experiments: Speed

Experiments: Speed



Prefixes of SYN



SYN

Final Remarks

Final Remarks

- Introduced the Combinatorial String Dissemination model which focuses on guaranteeing **privacy-utility trade-offs**.

Final Remarks

- Introduced the Combinatorial String Dissemination model which focuses on guaranteeing **privacy-utility trade-offs**.

- Defined three problems (**TFS**, **PFS**, and **MCSR**) to sanitize a string while preserving certain utility properties.

Final Remarks

- Introduced the Combinatorial String Dissemination model which focuses on guaranteeing **privacy-utility trade-offs**.
- Defined three problems (**TFS**, **PFS**, and **MCSR**) to sanitize a string while preserving certain utility properties.
- Developed methods (**TFS-ALGO**, **PFS-ALGO**, and **MCSR-ALGO**) for solving these problems.

Final Remarks

- Introduced the Combinatorial String Dissemination model which focuses on guaranteeing **privacy-utility trade-offs**.
- Defined three problems (**TFS**, **PFS**, and **MCSR**) to sanitize a string while preserving certain utility properties.
- Developed methods (**TFS-ALGO**, **PFS-ALGO**, and **MCSR-ALGO**) for solving these problems.
- Our experiments show that our methods are **effective** and **efficient**.

Final Remarks

- Introduced the Combinatorial String Dissemination model which focuses on guaranteeing **privacy-utility trade-offs**.
- Defined three problems (**TFS**, **PFS**, and **MCSR**) to sanitize a string while preserving certain utility properties.
- Developed methods (**TFS-ALGO**, **PFS-ALGO**, and **MCSR-ALGO**) for solving these problems.
- Our experiments show that our methods are **effective** and **efficient**.

Conference version: ECML/PKDD 2019

Final Remarks

- Introduced the Combinatorial String Dissemination model which focuses on guaranteeing **privacy-utility trade-offs**.
- Defined three problems (**TFS**, **PFS**, and **MCSR**) to sanitize a string while preserving certain utility properties.
- Developed methods (**TFS-ALGO**, **PFS-ALGO**, and **MCSR-ALGO**) for solving these problems.
- Our experiments show that our methods are **effective** and **efficient**.

Conference version: ECML/PKDD 2019
Full version: `arxiv.org/abs/1906.11030`