Distributed Privacy-Preserving Methods for Statistical Disclosure Control

Javier Herranz, Jordi Nin and Vicenç Torra

DPM 2009, St. Malo, 24/09/2009

UPC (Spain) LAAS-CNRS (France) IIIA-CSIC (Spain)

Outline

Outline

Definition

- A statistical data set X can be seen as a matrix with n rows (**records**) and V columns (**attributes**), where each row contains V attributes of an individual.
- Identifier attributes are removed (encrypted). Quasi-identifier attributes can be **confidential** or **non-confidential**.

	Non-Confidential			Confidential		
	age	...	ZIP	salary	...	#diseases
record 1	**	**	**	**	**	**
record 2	**	**	**	**	**	**
...	...	...	...	...	...	...
record n	**	**	**	**	**	**

Useful Data vs. Privacy Protection

- Some companies or institutions may be interested in obtaining statistical values related to the data in X.
- Releasing the data set X would compromise the privacy of the data.
- The solution is to release a **modified** data set $X' = \rho(X)$.
- **Goal:** X' must allow to obtain useful statistical information about X, whereas X' must protect as much as possible the privacy of the original data.
- These two aspects, privacy and utility, are in contradiction. Therefore, one must find a good **trade-off** between them.

How to Modify X ?

- Since the most statistically interesting information of $X = X_{nc}||X_c$ uses to be the confidential attributes, a very popular strategy is to modify only X_{nc}.
- Therefore, $X' = \rho(X_{nc})||X_c$, for some transformation (or **perturbation**) ρ applied to the non-confidential attributes.

How to Modify X ?

- Since the most statistically interesting information of $X = X_{nc}||X_c$ uses to be the confidential attributes, a very popular strategy is to modify only X_{nc}.
- Therefore, $X' = \rho(X_{nc})||X_c$, for some transformation (or **perturbation**) ρ applied to the non-confidential attributes.
- Some examples of perturbation methods ρ:
  - adding random **noise** to each entry,
  - **swapping** different entries of the same attribute,
  - **resampling**,
  - clustering techniques, like **microaggregation**,
  - we propose a new method: **rank shuffling**.

Outline

Database X Is Distributed

- Suppose the database X is not owned by a single party; instead, t users own disjoint parts of X:
  a set $\{P_1, \ldots, P_t\}$ of t users want to **jointly** compute $X' = \rho(X)$, where:
  - $X = X_1 \cup \ldots \cup X_t$,
  - X_i the secret input of user P_i,
  - no information on X_i is leaked in the protocol, other than what is deduced from the output X'.

Database X Is Distributed

- Suppose the database X is not owned by a single party; instead, t users own disjoint parts of X:
  a set $\{P_1, \ldots, P_t\}$ of t users want to **jointly** compute $X' = \rho(X)$, where:
  - $X = X_1 \cup \ldots \cup X_t$,
  - X_i the secret input of user P_i,
  - no information on X_i is leaked in the protocol, other than what is deduced from the output X'.

- The idea is to realize, in the **real world**, the following ideal functionality: a trusted third party (TTP) secretly receives X_i from each P_i, reconstructs the whole X, applies the perturbation ρ and publishes the result $X' = \rho(X)$.

Multiparty Computation

- This problem is a **particular case** of the general concept of **multiparty computation protocol**:
  a set $\{P_1, \ldots, P_t\}$ of t users want to **jointly** compute $y = f(x_1, \ldots, x_t)$, where:
  - x_i is the secret input of user P_i,
  - no information on x_i is leaked in the protocol, other than what is deduced from the output y.

Multiparty Computation

- This problem is a **particular case** of the general concept of **multiparty computation protocol**:
  a set $\{P_1, \ldots, P_t\}$ of t users want to **jointly** compute $y = f(x_1, \ldots, x_t)$, where:
  - x_i is the secret input of user P_i,
  - no information on x_i is leaked in the protocol, other than what is deduced from the output y.
- **Any** function f can be securely computed in this way [A. Yao, 1982].
- The generic solution is very inefficient; the goal is to find more efficient solutions for particular cases of f.

Outline

Swapping Methods

- The perturbation works **attribute by attribute**.
- A value of an attribute is swapped with a *close* value of the same attribute.

Swapping Methods

- The perturbation works **attribute by attribute**.
- A value of an attribute is swapped with a *close* value of the same attribute.

Example

Original, X			Protected, X'		
at_1	at_2	at_3	at_1'	at_2'	at_3
1	4	high	5	6	high
2	15	low	3	17	low
3	5	very low	2	8	very low
5	8	very high	1	5	very high
6	17	medium	8	15	medium
7	6	very high	9	4	very high
8	18	medium	6	16	medium
9	16	low	7	18	low

Distributed Swapping Methods Are Insecure

- A simple example with $t = 2$ users shows that one of them may easily identify the confidential and non-confidential attributes of the other user.
- This problem is **inherent** to swapping methods, even if the distributed version is ideally realized with a TTP.

Distributed Swapping Methods Are Insecure

- A simple example with $t = 2$ users shows that one of them may easily identify the confidential and non-confidential attributes of the other user.
- This problem is **inherent** to swapping methods, even if the distributed version is ideally realized with a TTP.

Example

Original, X			Protected, X'		
at_1	at_2	at_3	at_1'	at_2'	at_3
1	4	high	5	6	high
2	15	low	3	17	low
3	5	very low	2	8	very low
5	8	very high	1	5	very high
6	17	medium	8	15	medium
7	6	very high	9	4	very high
8	18	medium	6	16	medium
9	16	low	7	18	low

Outline

Rank Shuffling: The Protocol

Inputs: original dataset X with n records, window size p, window slide s

Rank Shuffling: The Protocol

Inputs: original dataset X with n records, window size p, window slide s

For each attribute at_j to be protected:

1. records of X are sorted in increasing order of the values x_{ij},

Rank Shuffling: The Protocol

Inputs: original dataset X with n records, window size p, window slide s

For each attribute at_j to be protected:

1. records of X are sorted in increasing order of the values x_{ij},
2. $f = 1, \quad \ell = p$

Rank Shuffling: The Protocol

Inputs: original dataset X with n records, window size p, window slide s

For each attribute at_j to be protected:

1. records of X are sorted in increasing order of the values x_{ij},
2. $f = 1, \quad \ell = p$
3. while $\ell \leq n$:
   - Random_Shuffle($x_{fj}, \ldots, x_{\ell j}$),

Rank Shuffling: The Protocol

Inputs: original dataset X with n records, window size p, window slide s

For each attribute at_j to be protected:

1. records of X are sorted in increasing order of the values x_{ij},
2. $f = 1, \quad \ell = p$
3. while $\ell \leq n$:
   - Random_Shuffle($x_{fj}, \ldots, x_{\ell j}$),
   - $f = f + s, \quad \ell = \ell + s.$

Rank Shuffling: an Example

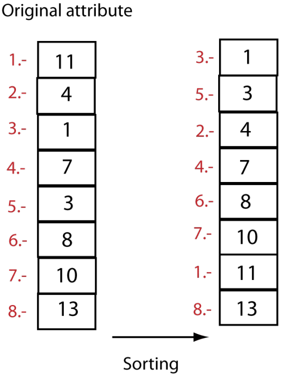
One attribute with $n = 8$ records, with $p = 4$ and $s = 2$.

Original attribute

1.-	11
2.-	4
3.-	1
4.-	7
5.-	3
6.-	8
7.-	10
8.-	13

Rank Shuffling: an Example

One attribute with $n = 8$ records, with $p = 4$ and $s = 2$.



Rank Shuffling: an Example

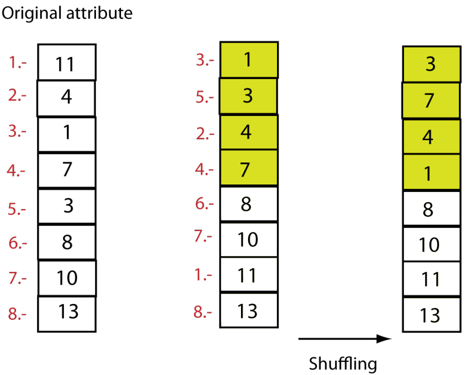
One attribute with $n = 8$ records, with $p = 4$ and $s = 2$.

Original attribute

1.-	11	3.-	1
2.-	4	5.-	3
3.-	1	2.-	4
4.-	7	4.-	7
5.-	3	6.-	8
6.-	8	7.-	10
7.-	10	1.-	11
8.-	13	8.-	13

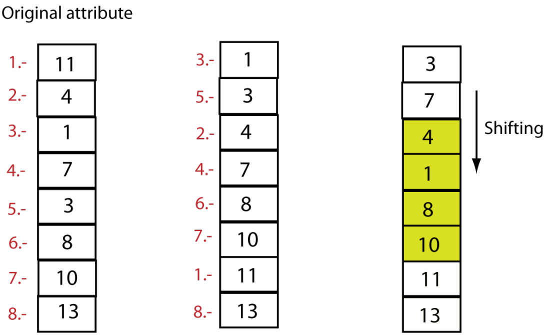
Rank Shuffling: an Example

One attribute with $n = 8$ records, with $p = 4$ and $s = 2$.



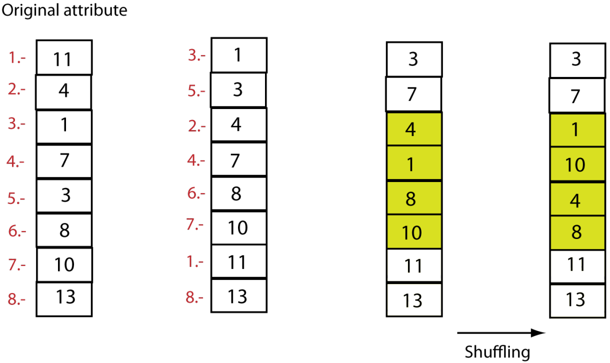
Rank Shuffling: an Example

One attribute with $n = 8$ records, with $p = 4$ and $s = 2$.



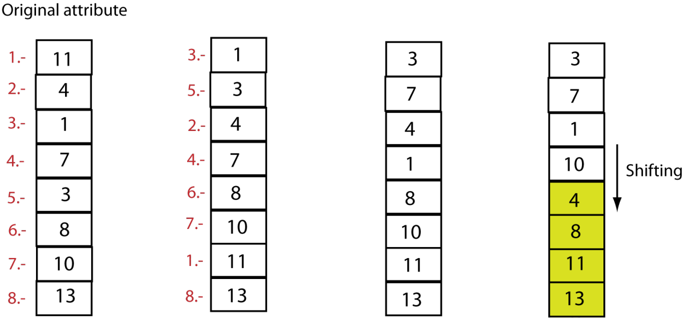
Rank Shuffling: an Example

One attribute with $n = 8$ records, with $p = 4$ and $s = 2$.



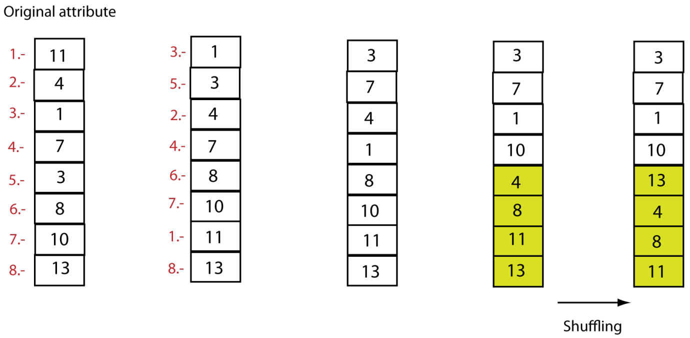
Rank Shuffling: an Example

One attribute with $n = 8$ records, with $p = 4$ and $s = 2$.



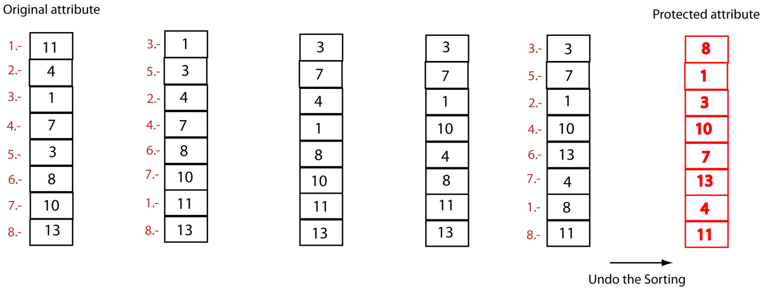
Rank Shuffling: an Example

One attribute with $n = 8$ records, with $p = 4$ and $s = 2$.



Rank Shuffling: an Example

One attribute with $n = 8$ records, with $p = 4$ and $s = 2$.



Rank Shuffling: Experimental Results

We have run Rank Shuffling on the Census dataset, using the software in `http://ppdm.iiia.csic.es`

Rank Shuffling: Experimental Results

We have run Rank Shuffling on the Census dataset, using the software in `http://ppdm.iiia.csic.es`

	IL	*DR*	*Score*	*Time* (sec.)
noise0.1	18.47	46.50	32.49	0.013
noise0.2	38.11	25.16	31.64	0.014
rs.5	30.78	14.90	22.84	0.47
rs.10	36.71	5.92	21.31	0.47
rs.15	37.57	4.20	20.88	0.42
resampling.2	29.84	84.61	58.21	0.50
resampling.4	21.95	90.71	53.72	0.82
rsshuffle.10-8	36.32	7.45	21.89	0.29
rsshuffle.25-20	35.85	4.67	20.26	0.28

Outline

Homomorphic Public Key Encryption

- Public key cryptography: a public key *pk* and a matching secret key *sk*.
- Encryption function $\varepsilon_{pk} : \mathcal{M} \times \mathcal{R} \rightarrow \mathcal{C}$.
- Decryption function $\mathcal{D}_{sk} : \mathcal{C} \rightarrow \mathcal{M}$.
- If the system is secure, $c = \varepsilon_{pk}(m)$ does not leak anything about m.

Homomorphic Public Key Encryption

- Public key cryptography: a public key *pk* and a matching secret key *sk*.
- Encryption function $\varepsilon_{pk} : \mathcal{M} \times \mathcal{R} \rightarrow \mathcal{C}$.
- Decryption function $\mathcal{D}_{sk} : \mathcal{C} \rightarrow \mathcal{M}$.
- If the system is secure, $c = \varepsilon_{pk}(m)$ does not leak anything about m.

Additive homomorphic property

$$\mathcal{D}_{sk}\left(\varepsilon_{pk}(m_1) \oplus \varepsilon_{pk}(m_2)\right) = m_1 + m_2,$$

for some operation $\oplus$ in the set of ciphertexts.

Threshold Decryption

- A trusted entity generates (sk, pk) and then splits sk into **shares**:

$$sk \longleftrightarrow \{sk_1, \ldots, sk_t\}$$

following a (k, t)-threshold *secret sharing scheme*, where $1 \leq k \leq t$.

- Each user P_i secretly holds the share sk_i.

Threshold Decryption

- A trusted entity generates (sk, pk) and then splits sk into **shares**:

$$sk \longleftrightarrow \{sk_1, \ldots, sk_t\}$$

  following a (k, t)-threshold *secret sharing scheme*, where $1 \leq k \leq t$.
- Each user P_i secretly holds the share sk_i.
- Given a ciphertext $c = \varepsilon_{pk}(m)$:
  - any $\geq k$ users **can** jointly decrypt and obtain m,
  - any $< k$ users cannot obtain any information on m.
- **Paillier**'s cryptosystem (1999) is additively homomorphic and allows threshold decryption.

Sub-protocol for Union

Input: each entity P_i has a set of elements $A_i = \{a_{i,1}, \ldots, a_{i,n_i}\}$
Output: encryptions of all these elements $\{\varepsilon_{pk}(a_{i,j})\}_{1 \leq i \leq t, 1 \leq j \leq n_i}$, in a random and unknown order.

Sub-protocol for Union

Input: each entity P_i has a set of elements $A_i = \{a_{i,1}, \ldots, a_{i,n_i}\}$
Output: encryptions of all these elements $\{\varepsilon_{pk}(a_{i,j})\}_{1 \leq i \leq t, 1 \leq j \leq n_i}$, in a random and unknown order.

- The goal is to hide which elements correspond to each entity.
- ε_{pk} must be additively homomorphic.
- **Idea:** each party re-encrypts, shuffles and sends the database to the following party.

Sub-protocol for Union

Input: each entity P_i has a set of elements $A_i = \{a_{i,1}, \ldots, a_{i,n_i}\}$
Output: encryptions of all these elements $\{\varepsilon_{pk}(a_{i,j})\}_{1 \leq i \leq t, 1 \leq j \leq n_i}$, in a random and unknown order.

- The goal is to hide which elements correspond to each entity.
- ε_{pk} must be additively homomorphic.
- **Idea:** each party re-encrypts, shuffles and sends the database to the following party.

We will denote an execution of this protocol as

$$C \leftarrow \mathtt{Union}(\{a_{i,j}\}_{1 \leq i \leq t, 1 \leq j \leq n_i})$$

Sub-protocol for Multiplication

- **Input:** $\varepsilon_{pk}(a)$ and $\varepsilon_{pk}(b)$ **Output:** $\varepsilon_{pk}(ab)$.
- We assume that ε_{pk} is additively homomorphic and allows $(t, t$-threshold decryption:
  - $\varepsilon_{pk}(a) \oplus \varepsilon_{pk}(b) = \varepsilon_{pk}(a+b)$, for any values a, b
  - each user P_i holds a share sk_i of the secret key sk ; decryption is possible if and only if all users cooperate.
- We will denote $\varepsilon_{pk}(ab) \leftarrow \texttt{Multip}(\varepsilon_{pk}(a), \varepsilon_{pk}(b))$.

  [Cramer-Damgård-Nielsen, 2001]

Sub-protocol for Bits

- Let $(a_{\ell-1}, \ldots, a_1, a_0) \in (\mathbb{Z}_2)^\ell$ be the **bit decomposition** of $a \in \mathbb{Z}_+$:

$$a = \sum_{0 \leq i \leq \ell-1} a_i 2^i.$$

- **Input:** $\varepsilon_{pk}(a)$ **Output:** $(\varepsilon_{pk}(a_{\ell-1}), \ldots, \varepsilon_{pk}(a_1), \varepsilon_{pk}(a_0))$.
- If ε_{pk} is Paillier's cryptosystem, then there are solutions for this task [Schoenmakers-Tuyls, 2006].
- We will denote $(\varepsilon_{pk}(a_{\ell-1}), \ldots, \varepsilon_{pk}(a_1), \varepsilon_{pk}(a_0)) \leftarrow \texttt{Bits}(\varepsilon_{pk}(a))$.

Sub-protocol for Comparison

- **Input:** $\varepsilon_{pk}(a)$ and $\varepsilon_{pk}(b)$.
- **Output:** $\begin{cases} \varepsilon_{pk}(1), & \text{if } a < b \\ \varepsilon_{pk}(0), & \text{if } a \geq b \end{cases}$

Sub-protocol for Comparison

- **Input:** $\varepsilon_{pk}(a)$ and $\varepsilon_{pk}(b)$.
- **Output:** $\begin{cases} \varepsilon_{pk}(1), & \text{if } a < b \\ \varepsilon_{pk}(0), & \text{if } a \geq b \end{cases}$
- **Idea:** $a \leftrightarrow (a_{\ell-1}, \ldots, a_1, a_0)$, $b \leftrightarrow (b_{\ell-1}, \ldots, b_1, b_0)$.
- *Privately* find the largest j such that $a_j \neq b_j$ (in other words, $a_j \text{ XOR } b_j = 1$). Note that $\varepsilon_{pk}(b_j)$ is the desired output.
- **Hint:** $e_i := a_i \text{ XOR } b_i = (a_i - b_i) \cdot (a_i - b_i)$

Sub-protocol for Comparison

- **Input:** $\varepsilon_{pk}(a)$ and $\varepsilon_{pk}(b)$.
- **Output:** $\begin{cases} \varepsilon_{pk}(1), & \text{if } a < b \\ \varepsilon_{pk}(0), & \text{if } a \geq b \end{cases}$
- **Idea:** $a \leftrightarrow (a_{\ell-1}, \ldots, a_1, a_0)$, $b \leftrightarrow (b_{\ell-1}, \ldots, b_1, b_0)$.
- *Privately* find the largest j such that $a_j \neq b_j$ (in other words, $a_j \text{ XOR } b_j = 1$). Note that $\varepsilon_{pk}(b_j)$ is the desired output.
- **Hint:** $e_i := a_i \text{ XOR } b_i = (a_i - b_i) \cdot (a_i - b_i)$

We will denote $\varepsilon_{pk}(b_j) \leftarrow \texttt{Compare}(\varepsilon_{pk}(a), \varepsilon_{pk}(b))$

Distributed Rank Shuffling: Setup

- The original database X, with V attributes, is horizontally partitioned among t entities $P_1, \ldots, P_t$.
- Let A_ℓ denote the set of indices of the records that belong to entity P_ℓ.
- Let pk be the public key of the employed threshold homomorphic encryption scheme ε (such as Paillier).
- Let p, s be the public parameters for rank shuffling: p is the window size, and s is the window slide.

Rank Shuffling: Reminder

Inputs: original dataset X with n records, window size p, window slide s

For each attribute at_j to be protected:

1. records of X are sorted in increasing order of the values x_{ij},
2. $f = 1, \quad \ell = p$
3. while $\ell \leq n$:
   - Random_Shuffle($x_{fj}, \ldots, x_{\ell j}$),
   - $f = f + s, \quad \ell = \ell + s$.

Distributed Rank Shuffling: the Protocol

1. P_ℓ computes, for each record $i \in A_\ell$, the tuple $(\{\varepsilon_{pk}(x_{ij})\}_{1 \le j \le V})$, that we denote as $\vec{c}_i = (c_{i1}, \ldots, c_{iV})$.

Distributed Rank Shuffling: the Protocol

1. P_ℓ computes, for each record $i \in A_\ell$, the tuple $(\{\varepsilon_{pk}(x_{ij})\}_{1 \le j \le V})$, that we denote as $\vec{c}_i = (c_{i1}, \ldots, c_{iV})$.
2. Run $C \leftarrow \texttt{Union}(\{\vec{x}_i\}_{1 \le \ell \le t, i \in A_\ell})$, where $\vec{x}_i = (x_{i1}, \ldots, x_{iV})$.

Distributed Rank Shuffling: the Protocol

1. P_ℓ computes, for each record $i \in A_\ell$, the tuple $(\{\varepsilon_{pk}(x_{ij})\}_{1 \le j \le V})$, that we denote as $\vec{c}_i = (c_{i1}, \ldots, c_{iV})$.
2. Run $C \leftarrow \texttt{Union}(\{\vec{x}_i\}_{1 \le \ell \le t, i \in A_\ell})$, where $\vec{x}_i = (x_{i1}, \ldots, x_{iV})$.
3. For each (non-confidential) attribute at_j to be protected:
    1. Making calls to `Compare`, sort the table C increasingly w.r.t. at_j.

Distributed Rank Shuffling: the Protocol

1. P_ℓ computes, for each record $i \in A_\ell$, the tuple $(\{\varepsilon_{pk}(x_{ij})\}_{1\leq j\leq V})$, that we denote as $\vec{c}_i = (c_{i1}, \ldots, c_{iV})$.
2. Run $C \leftarrow \texttt{Union}(\{\vec{x}_i\}_{1\leq \ell\leq t, i\in A_\ell})$, where $\vec{x}_i = (x_{i1}, \ldots, x_{iV})$.
3. For each (non-confidential) attribute at_j to be protected:
    1. Making calls to `Compare`, sort the table C increasingly w.r.t. at_j.
    2. Define $f = 0$ and $\ell = p$.

Distributed Rank Shuffling: the Protocol

1. P_ℓ computes, for each record $i \in A_\ell$, the tuple $(\{\varepsilon_{pk}(x_{ij})\}_{1 \le j \le V})$, that we denote as $\vec{c}_i = (c_{i1}, \ldots, c_{iV})$.
2. Run $C \leftarrow \texttt{Union}(\{\vec{x}_i\}_{1 \le \ell \le t, i \in A_\ell})$, where $\vec{x}_i = (x_{i1}, \ldots, x_{iV})$.
3. For each (non-confidential) attribute at_j to be protected:
    1. Making calls to `Compare`, sort the table C increasingly w.r.t. at_j.
    2. Define $f = 0$ and $\ell = p$.
    3. While $\ell \le n$ do:
        - (Iteratively) Re-randomize and permute the values $\{c_{fj}, \ldots, c_{\ell j}\}$.
        - $f = f + s, \qquad \ell = \ell + s.$

Distributed Rank Shuffling: the Protocol

1. P_ℓ computes, for each record $i \in A_\ell$, the tuple $(\{\varepsilon_{pk}(x_{ij})\}_{1\leq j\leq V})$, that we denote as $\vec{c}_i = (c_{i1}, \ldots, c_{iV})$.
2. Run $C \leftarrow \texttt{Union}(\{\vec{x}_i\}_{1\leq \ell\leq t, i\in A_\ell})$, where $\vec{x}_i = (x_{i1}, \ldots, x_{iV})$.
3. For each (non-confidential) attribute at_j to be protected:
    1. Making calls to `Compare`, sort the table C increasingly w.r.t. at_j.
    2. Define $f = 0$ and $\ell = p$.
    3. While $\ell \leq n$ do:
        - (Iteratively) Re-randomize and permute the values $\{c_{fj}, \ldots, c_{\ell j}\}$.
        - $f = f + s, \qquad \ell = \ell + s$.
4. Each P_ℓ re-randomizes and permutes the resulting vectors $\vec{c}_1, \ldots, \vec{c}_n$.

Distributed Rank Shuffling: the Protocol

1. P_ℓ computes, for each record $i \in A_\ell$, the tuple $(\{\varepsilon_{pk}(x_{ij})\}_{1 \leq j \leq V})$, that we denote as $\vec{c}_i = (c_{i1}, \ldots, c_{iV})$.
2. Run $C \leftarrow \texttt{Union}(\{\vec{x}_i\}_{1 \leq \ell \leq t, i \in A_\ell})$, where $\vec{x}_i = (x_{i1}, \ldots, x_{iV})$.
3. For each (non-confidential) attribute at_j to be protected:
   1. Making calls to `Compare`, sort the table C increasingly w.r.t. at_j.
   2. Define $f = 0$ and $\ell = p$.
   3. While $\ell \leq n$ do:
      - (Iteratively) Re-randomize and permute the values $\{c_{fj}, \ldots, c_{\ell j}\}$.
      - $f = f + s, \qquad \ell = \ell + s$.
4. Each P_ℓ re-randomizes and permutes the resulting vectors $\vec{c}_1, \ldots, \vec{c}_n$.
5. Decrypt jointly all the ciphertexts in the resulting table C.

Outline

Conclusions

- Situations where different entities want to compute a **global** protected dataset from their parts of original data can be easily found in **real life**.
- This motivates the problem of finding secure and distributed versions of the most popular SDC methods.

Conclusions

- Situations where different entities want to compute a **global** protected dataset from their parts of original data can be easily found in **real life**.
- This motivates the problem of finding secure and distributed versions of the most popular SDC methods.
- Some SDC do not admit a secure distributed version, like those in the **swapping family**.

Conclusions

- Situations where different entities want to compute a **global** protected dataset from their parts of original data can be easily found in **real life**.
- This motivates the problem of finding secure and distributed versions of the most popular SDC methods.
- Some SDC do not admit a secure distributed version, like those in the **swapping family**.
- For other SDC methods, distributed versions can be securely implemented by using secure **multiparty sub-protocols**: noise addition, resampling, rank shuffling.

Conclusions

- Situations where different entities want to compute a **global** protected dataset from their parts of original data can be easily found in **real life**.
- This motivates the problem of finding secure and distributed versions of the most popular SDC methods.
- Some SDC do not admit a secure distributed version, like those in the **swapping family**.
- For other SDC methods, distributed versions can be securely implemented by using secure **multiparty sub-protocols**: noise addition, resampling, rank shuffling.
- Open problem: distributed versions of SDC methods based on clustering, such as **microaggregation**.

Distributed Privacy-Preserving Methods for Statistical Disclosure Control

Javier Herranz, Jordi Nin and Vicenç Torra

DPM 2009, St. Malo, 24/09/2009

UPC (Spain) LAAS-CNRS (France) IIIA-CSIC (Spain)