Technical Report: Privacy-preserving Outsourcing of Association Rule Mining

Fosca Giannotti* Laks V.S. Lakshmanan† Anna Monreale ‡
Dino Pedreschi§ Hui (Wendy) Wang¶

Abstract

Spurred by developments such as cloud computing, there has been considerable recent interest in the paradigm of datamining-as-service. A company (data owner) lacking in expertise or computational resources can outsource its mining needs to a third party service provider (server). However, both the items in the outsourced database and the patterns of items that can be mined from the database, are considered as the private property of the corporation (data owner). To protect the corporate privacy, the data owner transforms its data and ships it to the server. The server sends extracted patterns to the owner in response to the latter's mining queries. The owner recovers the true patterns from the extracted patterns received. In this paper, we study the problem of outsourcing the association rule mining task within a *corporate privacy-preserving* framework. We propose an attack model based on background knowledge and devise two schemes, namely *Frugal* and *RobFrugal* , for privacy-preserving outsourced mining, based on the concept of k-anonymity. The protection against the privacy violation attack comes from ensuring that each transformed item (itemset) is indistinguishable, w.r.t. the attacker's background knowledge, from at least k-1 other transformed items (itemsets). We show that the owner can recover the true patterns as well as their support by maintaining a compact synopsis. Finally, we empirically demonstrate using comprehensive experiments on a very large and real transaction database, that our techniques are effective, scalable, and protect privacy.

1 Introduction

In this technical report are available all the proofs and details omitted in the work "Privacy-preserving Mining of Association Rules from Outsourced Transaction Databases", that we submitted to ICDE2010.

*ISTI-CNR, Pisa, Italy
†Univ. of British Columbia, Vancouver, Canada
‡University of Pisa, Italy
§University of Pisa, Italy
¶Stevens Inst. of Tech., New Jersey, USA

2 Encryption/Decryption Schemes Section

Definition 1 The *Frugal* encryption scheme consists in grouping together cipher items in D^* into groups of k adjacent items in the item support table in decreasing order of support, starting from the most frequent item e_1.

Lemma 1 For every non-monotone grouping G, there is a monotone grouping G', obtained by swapping items between groups, such that $||G'|| \leq ||G||$.

Proof: Suppose to the contrary that the grouping G is non-monotone and that by swapping items between groups we obtain either G' non-monotone or $||G'|| > ||G||$. This means that in $G = (G_1, \ldots, G_p)$ there exists a group G_i s.t. for some $e \in G_i$ we have $supp(e) \leq m_{i+1}$, where m_{i+1} is the maximum support (in D) of any item in group G_{i+1}. Denote by e' the item with support m_{i+1} and suppose of exchanging the item $e \in G_i$ with $e' \in G_{i+1}$. It is immediate to verify that in this case G' is monotone. So, if we show that $||G'|| \leq ||G||$ we obtain a contradiction. Consider the case of $G' = (G_1, \ldots, G_i, G_{i+1}, \ldots, G_p)$ where $G'_i = (e_1, \ldots, e')$ and $G'_{i+1} = (e, \ldots, e_s)$. This means that we have $supp(e) = m'_{i+1}$ and $m_{i+1} > m'_{i+1}$, therefore $\sum_{\forall e_j \in G'i+1} (m'_{i+1} - supp(e_j)) < \sum_{\forall e_j \in Gi+1} (m_{i+1} - supp(e_j))$. As a consequence $||G'|| \leq ||G||$. Hence, grouping G' is monotone and $||G'|| \leq ||G||$, which yields a contradiction.

Lemma 2 Let G^{frug} be the grouping obtained using the above procedure. Then G^{frug} is optimal, i.e., it has the least size among all groupings of cipher items into groups of size $\geq k$ each.

Proof: We show that any grouping method different from *Frugal* is not optimal. Let F be the grouping obtained using *Frugal*. Notice that F is a partition of the items $\mathcal{I}$. Suppose to the contrary that a grouping X, different from *Frugal*, is optimal. A grouping method different from *Frugal* generates a grouping X that can be either a non-monotone grouping or a monotone grouping. If X is non-monotone then by the Lemma 1 we show that X is not optimal. If X is monotone and not obtained with *Frugal* procedure then we can have two case:

- Let m_i be the maximum support of items in X_i and let $e_1, e_2 \in X_p$ be the item with support m_p and the item with the second largest value, respectively. Moving e_1 from X_p to X_{p-1} will change the size of the groping to $||X|| + (m_{p-1} - m_p) - (|X_p| - 1)(m_p - supp_D(e_2))$, where $(m_{p-1} - m_p) - (|X_p| - 1)(m_p - supp_D(e_2)) < 0$.
- Let m_i be the maximum support of items in X_i and let $e_1 \in X_{p-1}$ be the item with minimum support. Moving e_1 from X_{p-1} to X_p will change the size of the groping to $||X|| - (m_{p-1} - supp_D(e_1)) + |X_p|(supp_D(e_1) - m_p)$, where $(m_{p-1} - supp_D(e_1)) + |X_p|(supp_D(e_1) - m_p) > 0$.

In both cases it is possible to obtain a grouping with smaller size, hence, grouping X is not optimal, which yields a contradiction.

Theorem 1 Let D be a TDB and D^* its encryption obtained using the grouping G^{frug}. Then D^* is the smallest k-private TDB for D, i.e., its size $||D^*||$ is minimal among all k-private encryptions of D.

Proof: By Lemma 2 we have that G^{frug} is optimal, hence no grouping $X \neq G^{frug}$ has the least size among all groupings of cipher items into groups of size $\geq k$ each. Since $||D^*|| = ||D|| + ||G^{frug}||$ we conclude that $||D^*||$ is minimal.

2.1 Details on the Construction of fake transactions

It should be noted that given the frequency of cipher items in the noise column, any exact covering of these occurrences by means of a suitable set of transactions yields a correct realization of our encryption scheme. However, we aim at devising a method for arranging fake transactions that allows for a compact synopsis with a strong protection level.

Given a noise table specifying the noise $N(e)$ needed for each cipher item e, we generate the fake transactions as follows. First, we drop the rows with zero noise, corresponding to the most frequent items of each group or to other items with support equal to the maximum support of a group. Second, we sort the remaining rows in descending order of noise. Let $e'_1, \ldots, e'_m$ be the obtained ordering of (remaining) cipher items, with associated noise $N(e'_1), \ldots, N(e'_m)$. The following fake transactions are generated:

- $N(e'_1) - N(e'_2)$ instances of the transaction $\{e'_1\}$
- $N(e'_2) - N(e'_3)$ instances of the transaction $\{e'_1, e'_2\}$
- ...
- $N(e'_{m-1}) - N(e'_m)$ instances of the transaction $\{e'_1, \ldots, e'_{m-1}\}$
- $N(e'_m)$ instances of the transaction $\{e'_1, \ldots, e'_m\}$

Continuing the example, in the *Frugal* case we consider cipher items of non-zero noise in Table 1 (a).

(a) *Frugal* Scheme

Item	Support	Noise
e_2	5	0
e_4	3	2
e_5	2	0
e_1	1	1
e_3	1	1

Table 1: Noise table for $k = 2$

The following two fake transactions are generated: 1 instance of the transaction $\{e_4\}$ and 1 instance of the transaction $\{e_4, e_3, e_1\}$. In the *RobFrugal* case, we consider the cipher items with non-zero noise in Table 1 (b). The following 3 fake transactions are generated: 2 instances of the transaction $\{e_5, e_3, e_1\}$ and 1 instance of the transaction $\{e_5\}$.

It can be shown that this method yields a minimum number of different *types* of fake transactions that equals the number of cipher items with distinct noise (this number is 2 in both cases in the example). This observation yields a compact synopsis for the client, of the introduced fake transactions. As a final remark, we observe that fake transactions introduced by this method may be longer than any transactions in the original TDB D. Recall that the attack model only includes plain items and their exact support in D as the background knowledge of the attacker and not the transaction lengths in D. So, adding longer fake transactions technically does not constitute privacy breach. However, for added protection, we can consider shortening the lengths of the added fake transactions so that they are in line with the transaction lengths in D. In our running examples above, we obtain

in the *Frugal* case the fake transactions $\{e_4, e_3\}, \{e_1\}$, and $\{e_4\}$; in the *RobFrugal* we obtain $\{e_5, e_3\}$, 2 of $\{e_1\}$ and 1 instance of $\{e_5\}$. These transactions are of length either 1 or 2. We briefly illustrate the idea here. Let l be the length suggested by *Frugal*/*RobFrugal* for a fake transaction and let $l > l_{max}$, where l_{max} is the maximum length of a transaction in D. Then find the largest number $q : q \leq l_{max}$ and one of the following holds: (i) q divides l evenly, or (ii) l mod $q \approx q$, or (iii) l mod $q < \lfloor l/q \rfloor$. Here, we can take l mod $q \approx q$ to be l mod $q = q - 1$. If conditions (i) or (ii) hold, we simply split the fake transaction of length l into smaller ones of size q or $q - 1$. If condition (iii) holds, then we create $\lfloor l/q \rfloor$ transactions of size q. From the remaining set of l mod q items, we add one each to l mod q distinct transactions. So, we will have transactions of size q or $q + 1$. For example, suppose $l = 50$ and $l_{max} = 7$, the calculated q value equals to 7, i.e., the fake transaction of length 50 is split into 6 shorter ones of length 6, and 2 of length 7. In our experiments we studied the distribution of transaction length in both the original and encrypted TDBs, and observed that such distributions are very close. Figure 1 shows the distribution of transaction lengths before and after the encryption of the TDB $CoopProd$.

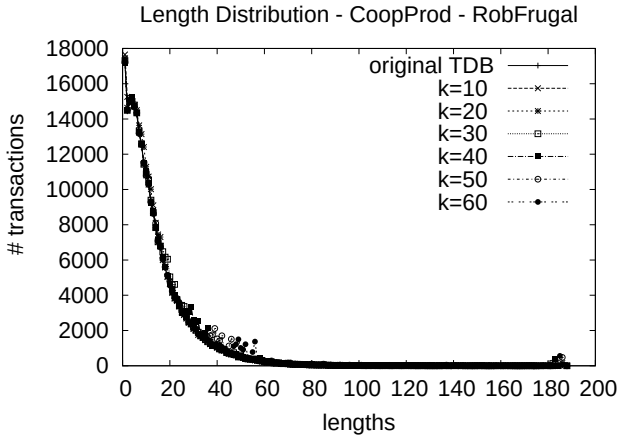

Figure 1: Distribution of transaction lengths for different k values

2.2 Implementing *RobFrugal* Efficiently

The encrypt/decrypt module is a "black box" to the users. We explain some implementation details of this black box. Note that all these details are hidden in the encrypt/decrypt module. Users need not concern themselves with them.

The key to achieve *RobFrugal* is that for any group G, efficiently check whether $supp(G) > 0$. For this purpose, the ED module makes use of a simpler version of FP-tree [1] in order to store the TDB in a compressed way; however, the ED module does not store transaction supports nor perform any mining. The data structure is composed of the well-known prefix tree structure and an item header table. In the prefix tree every node corresponds to an item in the database and contains three fields: the item name, the parent-link that is the link to the parent

node and the node-link that links to the next node in the tree carrying the same item-name or null is there is none. Every path represents the items that are in the same transaction. The paths share the same prefix sub-trees, if there is any. Each entry in the item header table consists of two fields: the item name and the head of the node-link. In Figure 2 we show and example of TDB stored by the prefix tree. Note that the prefix tree is constructed once by the ED module and is solely used for efficiently whether a given itemset has support > 0. The client can of course repeatedly issue mining queries to the server with various constraints on support threshold and item properties. The tree is constructed by two steps. First, scan the database once, collect the items, and sort them by their frequency in descending order. Then scan the database for the second time. During this scan, for each transaction, re-order the items by their sorted orders, and insert the re-ordered transaction into the prefix tree. In particular, for the item i that immediately follows item j in the transaction, let n_i and n_j be their corresponding nodes in the prefix tree. If n_j has no child of the same item name as n_i, then n_i is inserted as a new node under n_j. The procedure is repeated until all transactions are inserted into the tree. Then checking whether $supp(G) > 0$ is equivalent to checking whether G corresponds to a path in the tree and this can be done efficiently using this structure. In particular, given an itemset G first of all, we have to sort the items by their frequency in descending order, then we access to the header table using as key the last item e_j. Therefore, we access directly to the first occurrence of that item into the prefix tree, the node n_i. So, using the parent-link we visit the path reaching n_i, starting from it and going up to root. If this path contains all the items of G then $supp(G) > 0$. Otherwise, by using the link-node field of n_i it is possible to analyze others paths containing the last item e_j. If all the paths do not contain G then $supp(G) = 0$.

TID	Transactions	(ordered)Transactions
100	{f, a, c, m, p}	{f, c, a, m, p}
200	{a, b, c, f, m}	{f, c, a, b, m}
300	{b, f}	{f, b}
400	{b, c, p}	{c, b, p}
500	{a, f, c, e, d, p, m}	{f, c, a, m, p, d, e}

Item	freq.
f	4
c	4
a	3
b	3
m	3
p	3
d	1
e	1

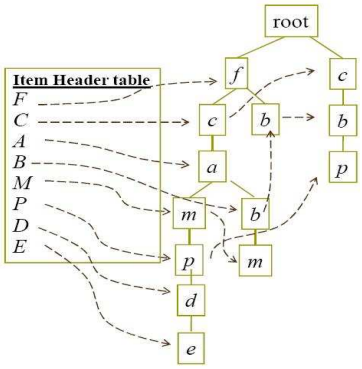

Figure 2: Prefix tree

3 Details on Complexity Analysis

In this section, we discuss time and space complexity of the *Frugal* and *RobFrugal* encryption schemes and associated decryption scheme. Recall that n is the number of (distinct) items in D and l_{max} is the maximum transaction length in D. Our next result justifies the name *Frugal* for the encryption scheme.

Theorem 2 The *Frugal* encryption scheme encodes the fake transactions using a synopsis of size $O(n)$, which can be built in $O(n\ log\ n)$ time, given the item support table of database D.

Proof: We show that each step of the *Frugal* encryption method is linear in n in space and costs $O(n\ log\ n)$ in time. We note that the *Grouping* step of procedure *Frugal* generates the groups of k adjacent items of the item support table, while generating the noise table. These two operations require to visit at most once all distinct items; therefore, this step requires $O(n)$ time. The second step, first of all, sorts the list of items with non null noise values in descending order of noise and then, creates hash tables in order to store the fake transactions efficiently. The cost of this operation is $O(n\ log\ n)$ for the sorting and $O(n - \frac{n}{k})$ for the creation of the hash tables. Indeed, we have to visit the list of items with non null noise values: by construction, at least the $\frac{n}{k}$ most frequent items of each group do not occur in this list. So, we conclude that the time complexity of the *Frugal* encryption is $O(n\ log\ n)$. In order to encode the fake transactions, *Frugal* uses the list of items with their support and noise value, requiring $O(n)$ space. Moreover, this method generates perfect hash tables containing all the items with non null noise values. Again, the collection of these items contains at most $n - \frac{n}{k}$ items because, for each group of k items, the item with highest support value has null noise value. Therefore, we conclude that the fake transactions are encoded using $O(n)$ space.

Theorem 3 The *RobFrugal* encryption scheme encodes the fake transactions in $O(n)$ storage and $O(n^2)$ time, given the item support table of database D and the prefix-tree representation of D.

Proof: The noise table and the hash tables, analogously to Thm. 2, require $O(n)$ space. Notice that *RobFrugal* in order to guarantee the k-private grouping has to assure that each k-group does not occur in D. In order to check this fact efficiently the ED module makes use of data structure described in Sec. 2.2 in order to represent D. The generation of fake transactions requires: (1) to generate the k-private grouping and (2) the creation of the hash tables. The phase (1) generates a grouping such that for each k-group G, $supp_D(G) = 0$. For this check this step uses the prefix tree as described in Sec. 2.2. The worst case requires to check the support of $O(n^2)$ k-groups, where n is the number of items. Given a k-group G, in order to check whether $supp_D(G) = 0$, at worst case all the paths, containing the last item e_j of G, have to be visited. This operation can be done as explained in Sec. 2.2 efficiently. In general, in the prefix tree the number of occurrences of a node with item-name e_j (denoted by $Occ(e_j)$) depends on both the support of the item and the position of the item in the sorted list of items. Given an item e_j in position x in the sorted list then e_j should occur in the tree at most 2^{x-1} times but it has to be considered also the support of this item in D. So, the number of occurrence of e_j is $Occ(e_j) = min\{supp_D(e_j), 2^{x-1}\}$. Considering that the item support distribution is described by a power law the max value of $Occ(e_j)$ is the maximum of the function $f = \frac{\alpha}{x^\beta} - 2^{x-1}$. Note that, a path in the prefix tree

represents a transaction therefore, for each path the number of nodes to be visited is l_{max}. Finally, the number of nodes to be visited to check whether $supp_D(G) = 0$ is $O(maxf * l_{max})$. Thus, the step (1) requires $O(maxf * l_{max} * n^2)$ in time. Clearly, $O(maxf * l_{max})$ in this kind of problem can be considered a constant, so the time required is $O(n^2)$.

The step (2) generates the hash tables and as showed in the Theorem 2 this requires $O(n\ log\ n)$ time. Clearly, the cost of the step (1) dominates the cost of the step (2). So we conclude that the generation of fake transactions, and so $RobFrugal$ requires $O(n^2)$ time.

Theorem 4 Given a cipher pattern E with its fake support from the server, the decryption procedure computes its actual support in $O(|E|)$ time.

Proof: In order to compute the actual support value of a given itemset E returned from the server, it is necessary to use the perfect hash tables that represent the fake transactions. First of all, the client by using any item $e \in E$ and the second-level hash function H selects the hash table HT containing e. Then, it selects the item $e_{max} \in E$ such that for each $e \in E\ h(e) < h(e_{max})$, where h is the perfect hash function for the hash table HT. In this way, the client can take the entry values of the hash table to compute the real support. To this purpose, a number of lookups equal to the number of items in the pattern is needed. The perfect hashing gives hash tables where the time to make a lookup is constant in the worst case. Therefore, the time complexity for the computation of the real support of a single pattern is $O(|E|)$.

3.1 Incremental Maintenance

We now consider incremental maintenance of the encrypted TDB. The ED module is responsible for this. We focus on batches of appends, which are very natural in data warehouses. Let D be an initial TDB and ΔD be a set of transactions that are appended. Let D^* be the original encrypted TDB. The ED module stores D as a prefix tree T. Let $syn(D, D^*)$ denote the compact synopsis stored by the ED module for encoding the generation of fake transactions in D^*. The server and client have the item support tables IST of D and IST^* of D^*.

Next, the new TDB ΔD arrives, together with its item support table IST_Δ. The following steps can be applied to obtain an incremental version of the ED module according to either of the *Frugal* or *RobFrugal* schemes:

1. The new transactions in ΔD are inserted into the prefix-tree T, obtaining a cumulative representation of $D \cup \Delta D$. Also, a cumulative item support table IST is constructed by adding the support of each item in IST^* and IST_Δ. In particular, for each item $e_i \in IST^*$ the support of e_i is added to the support of $e_i \in IST_\Delta$. Clearly, IST_Δ should both:

   a not contain some item belonging to IST^*;

   b contain some new items.

   In the case (a) the support of these items in the cumulative item support table IST is equal to the support of them in IST^*; while in the case (b) the support of these items in IST is equal to their support in IST_Δ. Note that, when the cumulative item support table IST is constructed the method keeps the order of the items in the IST^*. So, if an item belonging to IST^* is

in the position i, then in the cumulative item support table IST its position is i. When an item only belongs to the IST_Δ, then this item is appended to the list. Clearly, the balance of support in each group is now generally destroyed by the new item supports, and it is needed to add new fake transactions to restore the balance.

2. In the *Frugal* scheme, the pre-existing grouping is maintained, and the new synopsis for the new fake transactions F^* is constructed as usual. On the other hand, in the *RobFrugal* scheme, the old grouping is checked for robustness w.r.t. the overall prefix-tree T and the pre-existing synopsis, which is equivalent to checking against to $D^* \cup F^*$. If the check for robustness fails, than a new grouping is tried out with swapping, until a robust grouping is found. Then, the new synopsis for the new fake transactions is constructed as usual; notice that the new grouping is robust w.r.t. the new fake transactions by construction, as the most frequent item of each group does not occur in any fake transaction.
3. The ED module uses both old and new synopses to reconstruct the exact support of a pattern received from the server.

On the complexity side, we observe that the incremental encryption has a cost of $O(||\Delta D|| + n^2)$ time, namely the cost of updating the prefix tree with ΔD plus the cost of *RobFrugal*, where $O(n^2)$ is a very pessimistic upper bound. Besides the first term dominates the overall complexity for reasonably sized batches of appends. As a consequence, the incremental encryption has again a linear cost in the size of ΔD. Our method extends to the case when simultaneously, a new batch is appended and old batch is dropped; the method also works in the case when new items arrive or old items are dropped.

3.2 A toy example

We now present a toy example which shows how our incremental encryption procedure works when *RobFrugal* is used. Let D^* be the original encrypted TDB, obtained with $k = 3$. Consider its item support table IST^* and its synopsis representing the fake transactions, showed in Table 2 and Table 3, respectively.

Item	Support
e_1	10
e_2	10
e_3	10
e_4	5
e_5	5
e_6	5

Table 2: Item support table IST^* for toy example

Suppose another set of transactions ΔD arrives with the item support table IST_Δ (Table 4). Note that, IST_Δ does not contain the item e_6 while it contains the new item e_7.

The incremental procedure computes the cumulative item support table IST_{cum} (Table 5) as explained in Sec. 3.1.

	Table1
0	$\langle e_3, 0, 2\rangle$
1	$\langle e_6, 1, 1\rangle$
2	$\langle e_2, 0, 1\rangle$
3	$\langle e_5, 1, 0\rangle$

Table 3: Synopsis for D^*

Item	Support
e_1	2
e_2	21
e_7	2
e_3	3
e_4	10
e_5	12

Table 4: Item support table IST_Δ for toy example

Two cases are possible: 1) the grouping defined during the previous step is also robust for ΔD; 2) this grouping for ΔD is not robust.

In the case 1) we generate the fake transactions by using IST_{cum}. In the case 2) we have to generate a robust grouping for D^* and ΔD starting from IST_{cum}. Thus, we have to do queries to ΔD, D and fake transactions represented by the synopsis. We can do this using the prefix tree, representing D and ΔD, and the synopsis.

Suppose that the 3-groups $\{e_1\ e_2,\ e_3\}$ and $\{e_1,\ e_2,\ e_4\}$ occur in ΔD. Then the robust grouping, the noise table and the new synopsis are computed in order to represented the new set of fake transactions (Tables 6, 7). At this point the ED module has to use both old and new synopses (Tables 3, 7) to reconstruct the exact support of a pattern received from the server.

4 Proofs: Privacy Analysis Section

Theorem 5 For every cipher item $e \in \mathcal{E}$, let $Cand(e)$ be the corresponding candidate set computed by the above pruning procedure. Then every candidate set $Cand(e)$ is minimal. Furthermore, $Cand(e) = \{i' \mid supp_{D^*}(e') = supp_{D^*}(e)\}$, where i' is the true plain item corresponding to e'.

The proof of Theorem 5 is straighforward from the pruning procedure. For better understanding, we illustrate the theorem and the pruning procedure with an example.

Example 1 Consider a transaction database D containing items $i_1, ..., i_6$. Let e_j be the substitution cipher corresponding to i_j. Let the support distribution of items in D correspond to Figure 3, column "$supp_D(i)$". Suppose that the encrypted database D^* has a support distribution of cipher items matching the column "$supp_{D^*}(e)$". The last column, "Noise" corresponds to the additional

Item	Support
e_1	12
e_2	31
e_3	13
e_4	15
e_5	17
e_6	5
e_7	2

Table 5: Item support table IST_{cum} for toy example

Item	Support	Noise
e_2	31	0
e_5	17	14
e_1	12	19
e_4	15	0
e_3	13	7
e_6	5	10
e_7	2	13

Table 6: Noise table for $k = 3$

support of every cipher item that is caused by the addition of fake transactions. Then it is easy to verify that $ICand(e_1) = ICand(e_2) = \{i_1, i_2, i_3, i_4, i_5, i_6\}$, $ICand(e_3) = ICand(e_4) = \{i_3, i_4, i_5, i_6\}$, and $ICand(e_5) = ICand(e_6) = \{i_5, i_6\}$. Notice that cipher items with the same support in D^* have the same set of (initial) candidate sets. Now, after sorting cipher items on support, first, for e_5 and e_6, the attacker can infer that $Cand(e_5) = Cand(e_6) = \{i_5, i_6\}$. Second, the attacker removes i_5, i_6 and e_5, e_6 from further consideration. This removes i_5, i_6 from $ICand(e_3)$ and $ICand(e_4)$. Then the attacker concludes that $Cand(e_3) = Cand(e_4) = \{i_3, i_4\}$. Similarly, he concludes that $Cand(e_1) = Cand(e_2) = \{i_1, i_2\}$.

Assuming every candidate item is equally likely to be the true plain item corresponding to a given cipher item, we can show:

Item	$supp_D(i_j)$	$supp_{D^*}(e_j)$	Noise
i_1	100	100	0
i_2	67	100	33
i_3	40	40	0
i_4	33	40	7
i_5	20	20	0
i_6	8	20	12

Figure 3: Support of Plain and Cipher Items

	Table2
0	$\langle e_1, 5, 14\rangle$
1	$\langle e_5, 1, 13\rangle$
2	$\langle e_7, 3, 10\rangle$
3	$\langle e_6, 3, 7\rangle$
4	$\langle e_3, 7, 0\rangle$

Table 7: Synopsis for ΔD^*

Theorem 6 Let D be a transaction database and D^* the encrypted database produced by the *Frugal* scheme or the *RobFrugal* scheme. Then for every cipher item e, the probability of its crack is bounded by: $prob(e) \leq 1/k$, where k is a given parameter for item k-anonymity.

The correctness of Thm. 6 follows from the construction details of fake items for both *Frugal* and *RobFrugal*; by guaranteeing that every item is included in a candidate set of size at least k, the probability of cracking its cipher value is at most $1/k$. The theorem shows that the probability that an individual item is broken can always be controlled to be below a threshold chosen by the owner. By controlling the parameter k, the owner can control the crack probability of cipher items. The owner can choose a value for k by striking a balance between increased privacy and increased server side overhead for mining the encrypted database.

4.1 Set-based Attack

Theorem 7 Given a cipher itemset $E = \{e_1, e_2, \ldots, e_m\}$, let $C_1, ..., C_t$ be the collection of equivalence classes of E. Then the size of the candidate set of itemsets is $|Cand(E)| = \Pi_{i=1}^{t} \binom{|Cand(C_i)|}{|C_i|}$.

Proof: It is straightforward that E is a union of one or more equivalence classes. Construction of candidate itemsets from each equivalence class is independent of each other. Thus $|Cand(E)|$ equals to product of $\binom{|Cand(C_i)|}{|C_i|}$, the size of candidate itemsets constructed from the equivalent class C_i. Since $|Cand(C_i)| > |C_i|$ and $|Cand(C_i)| \geq k$, the result follows.

Theorem 8 Given the original transaction database D, let D_r^* be its encrypted version obtained using any robust grouping scheme. Then $\forall$ itemset E with non-zero support in D_r^*, the crack probability $prob(E) \leq 1/k$, where k is the given threshold for k-anonymity.

Proof: The key is to show that no cipher itemset can be complete under the *RobFrugal* scheme. Assume there is. Then E must be the union of one or more complete equivalence classes. In other words, every equivalence class in E has non-zero support in D_s^* . This contradicts the property ensured by the construction of *RobFrugal* . Thus there must exist at least one equivalence class that is not complete. Theorem 7 has shown that the bound of the candidate itemset for each incomplete equivalence class is at least k. Thus the size of candidate itemset for E must be at least k. The theorem follows.

References

[1] J. Han, J. Pei, Y. Yin, R. Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. Data Mining and Knowledge Discovery, 8, 5387, 2004.