Trust-but-Verify: Verifying Result Correctness of Outsourced Frequent Itemset Mining in Data-mining-as-a-service Paradigm

Boxiang Dong, Ruilin Liu, Hui (Wendy) Wang

Abstract—**Cloud computing is popularizing the computing paradigm in which data is outsourced to a third-party service provider (server) for data mining. Outsourcing, however, raises a serious security issue: how can the client of weak computational power verify that the server returned correct mining result? In this paper, we focus on the specific task of frequent itemset mining. We consider the server that is potentially untrusted and tries to escape from verification by using its prior knowledge of the outsourced data. We propose efficient probabilistic and deterministic verification approaches to check whether the server has returned correct and complete frequent itemsets. Our probabilistic approach can catch incorrect results with high probability, while our deterministic approach measures the result correctness with 100% certainty. We also design efficient verification methods for both cases that the data and the mining setup are updated. We demonstrate the effectiveness and efficiency of our methods using an extensive set of empirical results on real datasets.**

Index Terms—**Cloud computing, data mining as a service, security, result integrity verification.**

I. Introduction

The increasing ability to generate vast quantities of data presents technical challenges for efficient data mining. Outsourcing data mining computations to a third-party service provider (server) offers a cost-effective option, especially for data owners (clients) of limited resources. This introduces the data-mining-as-a-service ($DMaS$) paradigm. Cloud computing provides a natural solution for the $DMaS$ paradigm. A few active industry projects, for example, Google's Prediction APIs and Microsoft's Daytona project, provide cloud-based data mining as a service to users.

In this paper, we focus on *frequent itemset mining* as the outsourced data mining task. Informally, frequent itemsets refer to a set of data values (e.g., product items) whose number of co-occurrences exceeds a given threshold. Frequent itemset mining has been proven important in many applications such as market data analysis, networking data study, and human gene association study. Previous research has shown that frequent itemset mining can be computationally intensive, due to the huge search space that is exponential to data size as well as the possible explosive number of discovered frequent itemsets [15]. Therefore, for those clients of limited computational resources, outsourcing frequent itemset mining to computationally powerful service providers (e.g., the cloud) is a natural solution.

Although it is advantageous to achieve sophisticated analysis on tremendous volumes of data in a cost effective way, end users hesitate to place full trust in cloud computing. This raises serious security concerns. One of the main security issues is the *integrity* of the mining result. There are many possible reasons for the service provider to return incorrect answers [3]. For instance, the service provider would like to improve its revenue by computing with less resources while charging for more. Since sometimes the mining results are so critical that it is imperative to rule out errors during the computation, it is important to provide efficient mechanisms to verify the result integrity of outsourced data mining computations.

In this paper, we focus on the problem of verifying whether the server returned correct and complete frequent itemsets. By *correctness*, we mean that all itemsets returned by the server are frequent. By *completeness*, we mean that no frequent itemset is missing in the returned result.

[14] initialized the research on integrity verification of outsourced frequent itemset mining. Its basic idea is to insert some *fake items* that do not exist in the original dataset into the outsourced data. These fake items will construct a set of fake (in)frequent itemsets. Then by checking against the fake (in)frequent itemsets, the client can verify the correctness and completeness of the mining answer by the server. Their assumption is that the server has no background knowledge of the items in the outsourced datasets, thus it has equal probability to cheat on the fake and true itemsets. We argue that such assumption may not stand in practise, as the server may be able to possess prior knowledge of outsourced data (e.g., domain values of items and their frequency) from other sources [11]. Apparently such server can escape easily from the verification mechanism in [14] that is based on using fake items. Moreover, the server may also be aware of the knowledge of verification techniques, and tries to escape verification by utilizing such knowledge. Our goal is to design efficient and robust integrity verification methods to catch such server that may return incorrect and incomplete frequent itemsets. In particular, we make the following contributions. First, we design the *probabilistic approach* to catch mining result that does not meet the predefined correctness/completeness requirement with *high probability*. The key idea is to construct a set of (in)frequent itemsets from *real* items,

B. Dong, R. Liu and H.W. Wang are with the Department of Computer Science, Stevens Institute of Technology, Hoboken, NJ, USA.
Email: bdong@stevens.edu, rliu3@stevens.edu, Hui.Wang@stevens.edu

and use these (in)frequent itemsets as *evidence* to check the integrity of the server's mining results. We do not use any fake item as in [14] that does not exist in the original dataset to construct evidence (in)frequent itemsets.

Second, we design the *deterministic approach* to catch any incorrect/incomplete frequent itemset mining answer with *100% probability*. The key idea of our deterministic solution is to require the server to construct *cryptographic proofs* of the mining results. Both correctness and completeness of the mining results are measured against the proofs with 100% certainty.

Third, for both probabilistic and deterministic approaches, we provide efficient methods to deal with updates on both the outsourced data and the mining setup.

Last but not least, we complement our analytical results with extensive experiments evaluating the performance of our verification approaches. Our experimental results show that the probabilistic approach can achieve the desired verification guarantee with small overhead, while our deterministic approach provides higher security guarantee with overhead more than the probabilistic approach.

The paper is organized as follows. We discuss preliminaries in Section II. We present our probabilistic approach and deterministic approach in Section III and IV respectively. We discuss possible extension of our work in Section V. In Section VI, we evaluate the performance of our approach. We discuss related work in Section VII and conclude in Section VIII.

II. PRELIMINARIES

A. Frequent Itemset Mining

Given a transaction dataset D that consists of n transactions, let $\mathcal{I}$ be the set of unique items in D. The *support* of the itemset $I \subseteq \mathcal{I}$ (denoted as $sup_D(I)$) is the number of transactions in D that contain I. An itemset I is *frequent* if its support is no less than a support threshold min_{sup} [1]. Clearly the search space of all frequent itemsets is exponential to the number of items in D. The (in)frequent itemsets behave the following monotone property. For any given infrequent itemset I, any itemset I' s.t. $I \subseteq I'$ must be an infrequent itemset. Similarly, for any frequent itemset I, any itemset $I' \subseteq I$ must be a frequent itemset.

B. Outsourcing Setting

The data owner (client) outsources her dataset D, with the minimum support threshold min_{sup}, to the service provider (server). The server performs frequent itemset mining on the received dataset and returns the mining results to the client. We allow the client to use privacy-preserving frequent itemset mining algorithms [11], [6] to encrypt the dataset; in this case the server will return the accurate itemsets in encrypted format.

C. Untrusted Server

We consider two types of untrusted servers that may return erroneous answer: the *type-1* server that possesses the background knowledge of the outsourced dataset, including the domain of items and their frequency information, and the *type-2* server that is aware of the frequency distribution information of both items and transactions, as well as the details of the verification procedure. In this paper, we target at designing verification approaches to catch these two types of servers.

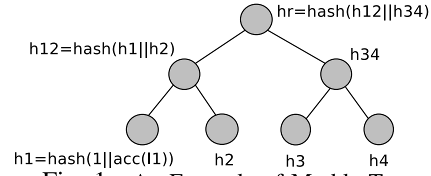


Fig. 1: An Example of Merkle Tree

D. Cryptographic Background

Bilinear pairings [19]. Let G_1 and G_2 be two cyclic multiplicative groups of order q for some large prime q. Let $e : G_1 \times G_1 \rightarrow G_2$ be a bilinear pairing with the following properties: (1) Bilinearity: there exists a bilinear map $e : G_1 \times G_1 \rightarrow G_2$ such that $e(P_1^a, P_2^b) = e(P_1, P_2)^{ab}$ for all $P_1, P_2 \in G_1$ and $a, b \in \mathbb{Z}$ of prime order q; (2) Non-degeneracy: $e(g, g) \neq 1$, i.e., not all pairs in $G_1 \times G_1$ are sent to the identity in G_2 by e; (3) Computability: there is an efficient algorithm to compute $e(P_1, P_2)$ for all $P_1, P_2 \in G_1$.

Merkle hash trees. A Merkle hash tree [20] is an authenticated data structure that allows data integrity verification. It is a binary tree $\mathcal{T}$ in which each leaf node l is assigned the value $h_l = hash(l||\mathcal{T}\ [l])$, while each non-leaf node with children a and b is assigned the value $h_l = hash(h_a||h_b)$, where $hash$ is a collision-resistant hash function. An important algorithm $\mathsf{MTproof}(v, \mathcal{T})$ is equipped with a Merkle hash tree $\mathcal{T}$, which outputs a proof showing that the digest value v is indeed the value stored in $\mathcal{T}$. In particular, let $path(i)$ be a list of nodes on the path from leaf i to the root and $sib(j)$ denote a sibling of node j in $\mathcal{T}$. Then $\mathsf{MTproof}()$ outputs the ordered list containing the hashes of the siblings $sib(j)$ of the nodes j in $path(i)$. For example, consider the Merkle tree $\mathcal{T}$ in Figure 1, and the value h_1. Then $\mathsf{MTproof}()$ returns $\{h_2, h_{34}\}$. The complexity of $\mathsf{MTproof}()$ is $O(log_2|\mathcal{T}|)$. How to use the Merkle tree for verification will be discussed in Section IV-B.

III. PROBABILISTIC APPROACH

The key idea of our methods is to construct a set of (in)frequent itemsets from real items, and use these (in)frequent itemsets as evidence to check the integrity of the server's mining result. We remove real items from the original dataset to construct artificial evidence infrequent itemsets (EIs), and insert copies of items that exist in the dataset to construct artificial evidence frequent items (EFs).

A. Verification Goal

Before discussing the detailed approaches, we formally define the correctness and completeness of the frequent itemset mining result. Let F be the real frequent itemsets in the outsourced database D, and F^S be the result returned by the server. We define the *precision* P of F^S as $P = \frac{|F \cap F^S|}{|F^S|}$ (i.e., the percentage of returned frequent itemsets that are correct), and the *recall* R of F^S as $R = \frac{|F \cap F^S|}{|F|}$ (i.e., the percentage of correct frequent itemsets that are returned). The aim of our probabilistic approaches is to catch any answer that does not meet the predefined precision/recall requirement with high probability. Formally, given a dataset D, let F^s be the set of frequent itemsets returned by the server. Let pr_R and pr_P be the probability to catch F^s of

recall $R \leq \alpha_1$ and precision $P \leq \alpha_2$, where $\alpha_1, \alpha_2 \in [0, 1]$ are given thresholds. We say a verification method M can verify (α_1, β_1)-*completeness* ((α_2, β_2)-*correctness*, resp.) if $pr_R \geq \beta_1$ ($pr_P \geq \beta_2$, resp.), where $\beta_1 \in [0, 1]$ ($\beta_2 \in [0, 1]$, resp.) is a given threshold. Our goal is to find a verification mechanism that can verify (α_1, β_1)-completeness and (α_2, β_2)-correctness.

B. Construction of Evidence Frequent Itemsets (EFs)

Our basic idea of completeness verification is that the client uses a set of frequent itemsets as the *evidence*, and checks whether the server misses any evidence frequent itemset in its returned result. If it does, the incomplete answer by the server is caught with 100% certainty. Otherwise, the client believes that the answer is complete with a probability. We have:

Theorem 3.1: Given the frequent itemsets F^S returned by the server with recall R, the probability pr_R of catching the incomplete answer with ℓ evidence frequent itemsets (EFs) is $pr_R = 1 - R^\ell$.

Clearly, to satisfy (α_1, β_1)-completeness (i.e., $pr_R \geq \beta_1$), it must be true that $\ell \geq \lceil log_{\alpha_1}(1 - \beta_1) \rceil$. Our analysis shows that with high completeness probability catching a server that fails to return even a small fraction of frequent itemsets does not need a large number of EFs. For instance, when $\alpha_1 = 0.95$ (i.e., 5% of frequent itemsets are not returned) and $\beta_1 = 0.95$, we only need 58 EFs. Apparently that the number of required EFs is independent from the size of the dataset as well as the number of real frequent itemsets. Therefore our verification approach is especially suitable for efficient verification for large datasets.

We propose the *MiniGraph* approach to construct EFs. Intuitively, we pick a set of item seeds for EF construction (Step 1), and find a set of real infrequent itemsets that contains at least one seed (Step 2 & 3). We take these infrequent itemsets as EFs and add fake transactions to make them artificially frequent (Step 4). Below we explain the details of the four steps of the *MiniGraph* approach.

Step 1: We pick the infrequent 1-itemset of the largest frequency as I_{seed}. The purpose of doing so is to reduce the number of artificial transactions added by Step 4. If there is no such 1-itemset, we pick the infrequent 2-itemset of the largest support. Note that I_{seed} is not infrequent anymore in the outsourced dataset after inserting artificial transactions.

Step 2: We find the transactions $D_{seed} \subseteq D$ that contain I_{seed}, and construct the *MiniGraph* G from D_{seed}. The root of G corresponds to I_{seed}. Each non-root node in G responds to a transaction in D_{seed}. There is an edge from node N_i to node N_j in G if the transaction that node N_j corresponds to is the *smallest superset* of the transaction of node N_i. Each node is labeled with an integer denoting the frequency of its corresponding transaction. Note that since I_{seed} is infrequent, the frequency of each node in *MiniGraph* does not exceed min_{sup}.

Step 3: We select ℓ nodes from the second level of G as EFs, where ℓ is the number of required EFs that satisfy (α_1, β_1)-completeness. The itemsets that the picked nodes correspond to are EFs. If there are not sufficient itemsets to select, we add the next infrequent 1-itemset of the largest frequency as another I_{seed}, and repeat Step 1 - 3, until we either find ℓ EFs or there is no infrequent 1-itemset left. If it cannot return enough EFs, we will continue the same procedure from the lower levels of G. It is easy to see that the maximum number of EFs that can be constructed by *MiniGraph* approach is $2^x - 2$, where x is the number of infrequent items in D. We assume that $2^x - 2 \geq \ell$, i.e., there are always sufficient EF candidates.

Step 4: For each picked EF: (1) we compute its support f as the total frequency of all its descendants and itself; (2) we compute the number of itemsets that need to be added as $(min_{sup} - f)$; and (3) we construct $(min_{sup} - f)$ artificial transactions, each being a copy of EF. These artificial transactions are inserted into D.

C. Construction of Evidence Infrequent Itemsets (EIs)

Similar to the completeness verification, our basic idea of correctness verification is that the client uses a set of infrequent itemsets as the *evidence*, and checks whether the server returns any evidence infrequent itemset. Next, we show how to measure the correctness probability guarantee. Similar to the completeness probability guarantee, we have:

Theorem 3.2: Given the frequent itemsets F^S returned by the server with precision P, the probability pr_P of catching the incorrect answer with ℓ EIs is $pr_P = 1 - P^\ell$.

To satisfy (α_2, β_2)-correctness (i.e., $pr_P \geq \beta_2$), it must satisfy that $\ell \geq \lceil log_{\alpha_2}(1 - \beta_2) \rceil$. As pr_P and pr_R (Theorem 3.1) are measured in the similar way, we have the same observation as for the completeness verification, which is that with high probability catching a server that returns even a small fraction of wrong frequent itemsets does not need large number of EIs. Furthermore, the number of required EIs is independent from the database size and the number of real frequent itemsets.

We propose the *lattice-based* approach to construct EIs. Specifically, we consider all infrequent 1-itemsets as EIs. If the number of such 1-itemsets is not sufficient to satisfy (α_2, β_2)-correctness, we transform a number of frequent 1-itemsets to be infrequent for additional EIs (Step 1). For those EIs that are frequent in the original dataset, we pick the minimum number of transactions (Step 2), and delete the minimum number of items (Step 3) from these transactions to make all EIs infrequent. Next, we explain the details of the lattice-based approach.

Step 1: Pick Evidence Infrequent Itemsets (EIs). First, we first all infrequent 1-itemsets. We exclude the I_{seed} used for EF construction from the set of 1-itemset candidates, so that I_{seed} and all its supersets will not be considered as the EI candidates. This ensures that no itemset will be required to be EI and EF at the same time. Let x be the number of infrequent 1-itemsets. These x infrequent 1-itemsets are inserted into the evidence repository $\mathcal{R}$. If $x > \ell$, where ℓ is the number of EIs required to satisfy (α_2, β_2)-correctness (i.e., there are sufficient number of EIs), we terminate construction of EIs; otherwise, we continue. Let h be the minimal value to make $\binom{m-x}{h} \geq \ell - x$, where m is the number of unique items in D, and x is the number of infrequent items in

D. Also let k be the minimal value to make $\binom{k}{h} \geq \ell - x$. We pick the first k *frequent* 1-itemsets S_{1item} from the sorted list of items. We construct all h-itemset candidates S_{hitem} that contain h items from S_{1item}, and insert those h-itemset candidates that have non-zero support into $\mathcal{R}$. If $|\mathcal{R}| < \ell$ (i.e., there is insufficient number of EIs), we pick the next k' sorted items S'_{1item} after S_{1item} in the sorted list, where $\binom{k+k'}{h} - \binom{k}{h} > (\ell - |\mathcal{R}|)$. We update $S_{1item} = S_{1item} \cup S'_{1item}$ as well as $k = k + k'$, and repeat Step 1.4, until there are at least ℓ h-itemsets of non-zero supports. If all unique items are inserted into S_{1item} but there are fewer than ℓ EIs, we increase h by 1, and repeat the procedure, until $\mathcal{R}$ has ℓ EIs.

For transaction datasets that have large number of infrequent 1-itemsets, the algorithm terminates early without the need to remove any item. It has been shown [4] that in real-life transaction datasets, the item support distribution (as well as the itemset support distribution) follows a power-law distribution: it is highly likely that there are large number of infrequent 1-itemsets. This is advantageous as in practice, the correctness verification can be achieved without changing the database.

Step 2: Pick Transactions for Item Removal. Out of ℓ EIs picked by Step 1, some of them are infrequent in D and some are indeed frequent. We do not need to take any action on the real infrequent itemsets RI. We aim at transforming those frequent EIs into artificial infrequent EIs (notated as AI). To achieve this, we pick a set of transactions $D' \subseteq D$, so that for each frequent itemset $I \in AI$, $sup_{D'}(I) \geq sup_D(I) - min_{sup} + 1$. In other words, for each I, its support will be decreased to at most $min_{sup} - 1$ after their instances in D' are removed. We aim at finding the minimal set of such D'.

To help pick D', we construct the *transaction matrix* M of D. M is a $u \times n$ binary matrix, where u is the number of itemsets in AI, and n is the number of transactions in D. $M[i, j]$ = 1 means that the frequent itemset I_i appears in transaction T_j; otherwise, $M[i, j]$ = 0. We also define a n-length binary vector V, in which $V[i]$ = 1 means that the i-th transaction is picked. Let $A = M \times V$. Apparently $A[i]$ equals the number of picked transactions that contain the i-th frequent itemsets in AI. Then the problem of finding the transactions D' for transforming AI to be infrequent is equivalent to finding a n-length binary vector V such that $\forall I_i \in AI$, $A[i] \geq sup_D(I_i) - min_{sup}$, where $A = M \times V$. We initialize A as:

$$A = \begin{pmatrix} sup_D(AI_1) - min_{sup} + 1 \\ \cdots \\ sup_D(AI_u) - min_{sup} + 1 \end{pmatrix}$$

Then we calculate V such that: (1) $M \times V = A$, and (2) V is a binary vector. To satisfy (1), we calculate $V = (M^T M)^{-1} M^T A$. If V is a binary vector, we return V. Otherwise, we pick the smallest value in A, and increase it by 1 (i.e. allow more instances to be removed). If there are multiple entries in A of the same values, we break the tie randomly. Whenever we update A, we increment the *relaxed cost* (initialized as 0) by 1. We repeat the calculation of V, until either there is a V to be returned or the amount of relaxed cost exceeds a given threshold.

Step 3: Pick Item Instances for Removal. Given the transactions D' obtained by Step 2, we need to decide which items in D' will be removed. We aim at minimizing the total number of removed items. To address this issue, we put high priority on removing the items that are shared among patterns in AI. Therefore, we do the following to pick the items. First, for each unique item in AI, we count its frequency in AI. Second, we sort the items by their frequency in descending order. Third, we construct the *item matrix* IM of AI. IM is a $u \times y$ binary matrix, where u is the number of itemsets in AI, and y is the number of unique items in AI. In the matrix, $IM[i, j]$ = 1 means that the frequent itemset I_i contains the item i_j; otherwise, $IM[i, j]$ = 0. Then we repeat the following procedure on IM. We add the item that corresponds to the first column of IM to the output, update IM by removing all rows i such that $IM[i, 1] = 1$ (i.e., all patterns that contain the item of the largest support), and re-sort the columns in the updated IM by their sum in descending order. We repeat until IM becomes empty. The sequence of the removed columns outputs the items to be picked for removal.

After item removal, by following the property of the downward closure of infrequentness, the client adds all descendant itemsets of the EIs picked by Step 1 that are of non-zero support into the evidence repository $\mathcal{R}$. Furthermore, we are aware that changing frequent itemsets to be infrequent will modify all of its frequent descendants to be infrequent. This will lead to incomplete frequent itemsets even when the server is honest. To solve this problem, for each descendant itemset of AI that is added to $\mathcal{R}$, we count its support and mark it as *recoverable* if it is frequent. Those recoverable itemsets are maintained locally by the client and will be added back to the mining result from the server. Finding recoverable frequent itemsets completes the construction of EIs.

D. Robustness Analysis

In this section, we analyze the robustness of our verification approach based on EFs and EIs.

Robustness of EFs. The type-1 server may compare the frequency of items between its background knowledge and that collected from the outsourced D'. It may notice that some items that are infrequent in D turn to be frequent in D'. Let these items be J. The server may choose not to cheat over the itemsets that contain any item in J. Note that J is a superset of all I_{seed} items that are used for EF construction. Let pr be the probability that the attacker can escape verification based on EFs constructed from one specific I_{seed}. Let j_1 and j_2 denote the number of I_{seed} items and items other than I_{seed} in J respectively. Then $pr = \frac{j_1}{j_1 + j_2}$. Thus the probability that the attacker can escape verification based on EFs constructed from j_1 I_{seed} items equals $Pr = 1 - (1 - pr)^{j_1}$. The client may require γ-*security*, i.e., the probability Pr that the attacker can escape the EF-based verification is no greater than γ. To ensure γ-*security*, we need to add artificial transactions to convert $j_2 = \lceil \frac{j_1}{1-(1-\gamma)^{\frac{1}{j_1}}} - j_1 \rceil$ infrequent 1-itemsets, besides I_{seed}, to be artificially frequent in D'.

The type-2 server is aware that the infrequent 1-itemset(s) with the largest frequency are always picked as I_{seed} items. Therefore, the probability that the server can identify those I_{seed} items (and thus escape the verification based on EFs constructed from these items) is 100%. To fix this issue, instead of picking I_{seed} from infrequent items in descending order of frequency, the client has to pick I_{seed} from the infrequent items randomly.

Robustness of EIs. The type-1 server may observe that the frequency drop of some items in the outsourced data and may decide not to cheat on EIs that contain these items. Let O be those items. Indeed, these items are those we choose to remove when constructing EIs. In this case, those EIs that contain any item of O would not be able to catch the server. Only $\binom{k-o}{h}$ EIs that do not contain any of O are still effective, where o is the number of items in O, and k is number of unique frequent items we pick to build h-itemsets as EIs. To fix this issue, we may require that o satisfies $\binom{k-o}{h} \geq \ell - x$, where ℓ is the number of EIs required to satisfy (α_2, β_2)-correctness, and x is the number of infrequent items picked as EIs.

The type-2 server understands how EIs are constructed. Thus, it can infer that all infrequent 1-itemsets must be EIs, and thus reduce the catching probability from $(1 - Pr^{|EI|})$ to $(1 - Pr^{|AI|})$, where AI are the set of EIs that are indeed frequent in D. To fix this problem, we require that $|AI| \geq \ell$. But it is likely that we can not find sufficient number of AIs. In this case, the probabilistic approach fails to provide the required correctness guarantee.

E. Complexity Analysis

In this section, we theoretically analyze the time and space complexity of our EF and EI construction.

1) EF construction: The time and space complexity of the *MiniGraph* approach are $O(|D|)$ and $O(|G|)$ respectively, where G is the constructed *MiniGraph*. Normally $|G|$ is a small portion of $|D|$. Compared with the complexity $O(2^m)$ of frequent itemset mining, where m is the number of items in D, our verification preparation approach has cheaper overhead than mining locally.

2) EI construction: The complexity of the lattice-based approach is $O(|EI||D|)$. The complexity of removing item instances is $O(y|AI|)$, where y is the number of unique items in AI. Since $|AI| \leq |EI|$ and $y << |D|$, the complexity of verification preparation for correctness verification is $O(|EI||D|)$.

F. Update Operations

The client may need to update her data from time to time and/or want to continuously execute frequent itemset mining over the outsourced dataset with different min_{sup} values. Let ΔD be the update over dataset D, and Δmin_{sup} the change in min_{sup}. For simplicity, we only consider insertion-only and deletion-only operations. We use $|\Delta D|$ to denote the size of ΔD. When ΔD corresponds to a sequence of deletions, we still consider $|\Delta D|$ as a positive value, which equals the number of deleted transactions. Δmin_{sup} could be either positive or negative integer values. Next, we discuss how to adapt our verification method to deal with the update on the dataset, as well as the update on min_{sup}.

When $\Delta min_{sup} > 0$. First, we discuss EFs update. The client has to ensure that the EFs are still frequent with regard to the new min_{sup}. Therefore, for each itemset $I \in EF$, if $\Delta f(I) = \Delta min_{sup} - sup_{\Delta D}(I) > 0$, the client inserts $\Delta f(I)$ artificial copies of I to D. Otherwise, no update is needed on I.

Regarding EIs update, we will ensure the infrequentness of EIs after update. If ΔD is a sequence of insertions and $|\Delta D| \geq \Delta min_{sup}$, the client applies Step 2 and 3 in Section III-C on ΔD to make sure that ΔD does not change the infrequentness of EIs. Specifically, the matrix M is constructed from ΔD, while A is constructed as

$$A = \begin{pmatrix} sup_{\Delta D}(EI_1) - \Delta min_{sup} + 1 \\ \cdots \\ sup_{\Delta D}(EI_u) - \Delta min_{sup} + 1 \end{pmatrix}.$$

Otherwise, no update on EIs is needed.

When $\Delta min_{sup} \leq 0$. First, we discuss EFs update. When ΔD is constructed by deletion-only transactions and $|\Delta D| + \Delta min_{sup} > 0$, it is likely that a EF may turn to be infrequent. Therefore, for each $I \in EF$, the client evaluates $\Delta f(I) = sup_{\Delta D}(I) + \Delta min_{sup}$. If $\Delta f(I) < 0$, the client inserts $|\Delta f(I)|$ copies of I to ΔD. Otherwise, there is no update on I.

Regarding EIs update, EIs may become frequent for the smaller min_{sup} value. Therefore, the client checks $sup_{D'}(I)$ for each $I \in EI$, where D' is the dataset after update. Let $f_{max} = max\{sup_{D'}(I) | I \in EI\}$. If $f_{max} \geq min_{sup} + \Delta min_{sup}$, the client applies the Step 2 and 3 in Section III-C on D', with M constructed from D', and A is constructed as following:

$$A = \begin{pmatrix} sup_{D'}(EI_1) - (min_{sup} + \Delta min_{sup}) + 1 \\ \cdots \\ sup_{D'}(EI_u) - (min_{sup} + \Delta min_{sup}) + 1 \end{pmatrix}.$$

Complexity Analysis. The time complexity of updates on EFs is $O(|\Delta D|)$. The time complexity of update on EIs can be $O(|\Delta D|)$ (when $\Delta min_{sup} > 0$) or $O(|D'|)$ (otherwise).

G. Post-Processing

Removal of artificial frequent itemsets is straightforward. As the client is aware of I_{seed} that is contained in all EFs, it needs to remove all the returned frequent itemsets that contain I_{seed}. Besides, we recover the original frequency of those itemsets that are contained by EF. Specifically, for any itemset I which is a subset of at least one EF, $sup_D(I) = sup_{D'}(I) - \sum_{I \subset I',\, I' \in EF}(sup_{D'}(I') - sup_D(I'))$. In this way, we eliminate those originally infrequent itemsets from the returned result.

To recover missing real frequent itemsets, the client maintains locally all AIs and their recoverable descendants when it constructs EIs. During post-processing, the client adds these AIs and the recoverable descendants back to F^S as frequent itemsets.

IV. Deterministic Approach

Our deterministic approach is based on an efficient authenticated data structure, which is built upon standard Merkle trees and bilinear-map accumulators. It enables

the proof-based verification. We optimize the verification algorithm by reducing the number of proofs for both correctness and completeness verification. We show that a small number of proofs is sufficient to verify the correctness and completeness of a large set of frequent itemsets.

A. Set Intersection Verification Protocol

In this paper, we adapt the set intersection verification protocol [21] to our problem. Formally, given a collection sets $\mathcal{S} = \{S_1, \ldots, S_v\}$, $E = S_1 \cap S_2 \cap \cdots \cap S_v$ is the *correct intersection* of $\mathcal{S}$ if and only if:

- $E \subseteq S_1 \wedge \cdots \wedge \subseteq S_v$ (subset condition);
- $(S_1 - E) \cap \cdots \cap (S_v - E) = \emptyset$ (completeness condition).

Here the completeness condition is on the set intersection. To avoid confusion with our completeness verification of frequent itemsets, in the remainder of the paper, we use the term *intersection completeness* for the completeness condition on the set intersection.

Papamanthou et al. proposed a set intersection verification protocol to verify that E is the correct intersection of $\mathcal{S}$ [21]. Next, we explain the protocol details briefly.

Proof construction by the server. Given an intersection result $E = \{e_1, \ldots, e_\delta\}$, to prove that $E = S_1 \cap \cdots \cap S_v$, the set intersection verification protocol requires the server to construct its proof $\Pi(E)$ that consists of four parts: (1) an encoding of the result (as a polynomial) as the coefficients $\mathcal{B} = \{b_\delta, b_{\delta-1}, \cdots, b_0\}$ of the polynomial $(s + e_1)(s + e_2) \cdots (s + e_\delta)$, where s is a randomly chosen value by the client and kept as secret; (2) a set of *accumulation values* $\mathcal{A} = \{acc(S_j) | \forall S_j \in \mathcal{S}\}$ which can be used to verify that the proof is indeed computed from the original dataset D; (3) the *subset witness* $\mathcal{W} = \{W_j | \forall S_j \in \mathcal{S}\}$ as the proof of the subset condition; and (4) the *intersection completeness witness* $\mathcal{C} = \{C_j | \forall S_j \in \mathcal{S}\}$ as the proof of the intersection completeness condition. The total complexity of proof construction is $O(Blog^3 B + m^\epsilon logm)$, where $B = \sum_{j=1}^{v} |S_j|$, m is the number of leaves of the Merkle tree of the dataset D, and $\epsilon \in (0, 1)$ decides the number of levels of the Merkle tree (as $\lceil 1/\epsilon \rceil$). More details of the Merkle tree will be present in Section IV-B.

Correctness verification by the client. After receiving $\Pi(E) = \{\mathcal{B}, \mathcal{A}, \mathcal{W}, \mathcal{C}\}$ together with E, the client verifies the following: (1) whether the coefficients $\mathcal{B}$ are computed correctly by the server; (2) whether any given accumulation value in $\mathcal{A}$ is indeed calculated from the original dataset; (3) whether E satisfies the subset condition by using $\mathcal{W}$; and (4) whether E satisfies the intersection completeness condition by using $\mathcal{C}$. We omit the details of proof construction and verification due to limited space. Readers can refer to [21] for more details.

B. Basic Solution

In this section, we present our basic verification approach. We first present in Section IV-B1 the authenticated data structure that we will design. Then in Section IV-B2, we show how to adapt the set operation verification protocol to the problem of frequent itemset mining by utilizing the authenticated data structure. In Section IV-B3, we discuss the robustness and weakness of the basic approach.

TID	Transactions
1	$\{i_1, i_2, i_3, i_4\}$
2	$\{i_1, i_2, i_4\}$
3	$\{i_2, i_4\}$
4	$\{i_1, i_2, i_3\}$
5	$\{i_1, i_3, i_4\}$
6	$\{i_4\}$
7	$\{i_2, i_3\}$
8	$\{i_1, i_2\}$

(a) Transaction dataset D

Item	Inverted List
i_1	1, 2, 4, 5, 8
i_2	1, 2, 3, 4, 7, 8
i_3	1, 4, 5, 7
i_4	1, 2, 3, 5, 6

(b) Item-based Inverted index E^I

Fig. 2: An example of the dataset and its inverted index

1) Authenticated Data Structure: Before sending the dataset D to the server, the client constructs an authenticated data structure. Before we discuss the details of the authenticated data structure, we first discuss the *item-based inverted index*. The authenticated data structure will be constructed from the inverted index. In particular, given a dataset D, its *item-based inverted index* E^I consists of a set of inverted lists $\{L_1, L_2, \ldots, L_m\}$, where m is the number of unique items in D. Each inverted list $L_i \in E^I$ corresponds to the item I_i in D, and maintains the index of transactions that contains the item I_i. As an example, consider the transaction dataset D shown in Figure 2 (a), Figure 2 (b) shows its item-based inverted index of D.

Now we are ready to discuss how to construct the authenticated data structure. We use the Merkle hash tree $\mathcal{T}$ of the inverted index as our authenticated data structure. In particular, the client picks a random value $s \in \mathbb{Z}$ which is kept secret. Then, for each leaf l_j of $\mathcal{T}$ that corresponds to the j-th inverted list L_j in E^I, the client constructs $acc(l_j) = g^{\prod_{x \in L_j}(s+x)}$, where g is a generator of the group G_1 from an instance of bilinear pairing parameters. Then the client applies a collision-resistant hash function $hash(\cdot)$ recursively over the nodes of $\mathcal{T}$. Each leaf l_j of $\mathcal{T}$ is assigned the value $h_j = hash(v_1 || \ldots || v_w || acc(l_j))$, where $v_1, \ldots, v_w$ are the values in the j-th inverted list L_j that l_j corresponds to, while each internal node v with children a and b is assigned to $h_v = hash(h_a || h_b)$. The root of the tree is signed to produce signature $sig(E^I)$. The client sends $\mathcal{T}$ to the server with D, and keeps $sig(E^I)$ locally. The complexity of constructing a Merkle tree of level $\lceil 1/\epsilon \rceil$ levels and m leaves is $O(m + H)$, where $\epsilon \in (0, 1)$ is a user-specified constant, and $H = \sum_{j=1}^{m} |l_j|$.

At this point, the client can find all frequent 1-itemsets and infrequent 1-itemsets from the inverted index. It maintains such information for later verification.

2) Verification Procedure: Before outsourcing the dataset D to the server, the client constructs the item-based inverted index E^I of D, as well as the Merkle hash tree $\mathcal{T}$ of E^I. The client keeps the hash value of the root element of $\mathcal{T}$, and sends D and $\mathcal{T}$ to the server.

Proof construction by the server. After the server receives D and $\mathcal{T}$ from the client, the server runs frequent itemset mining on D and discovers a set of frequent itemsets. Before the server sends its result F^S to the client, it constructs *proofs* for correctness and completeness verification. In particular, for each itemset $I \in F^S$, the server constructs a proof $\Pi(I)$ [21]. Similarly, for each itemset $I \notin F^S$, the server constructs a proof too. The proof of the itemset

$I \in F^S$ will be used for the correctness verification, while the proof of the itemset $I \notin F^S$ will be used for the completeness verification,

Result verification by the client. After receiving the frequent itemsets F^S, for each itemset $I \in F^S$ and its proof $\Pi(I) = \{\mathcal{B}, \mathcal{A}, \mathcal{W}, \mathcal{C}\}$, the client launches the following steps to verify that I is frequent (i.e., I is contained in at least min_{sup} transactions T^I): (1) It checks the size of T^I by checking whether $|\mathcal{B}| \geq min_{sup} + 1$; (2) It verifies the validity of $\mathcal{B} \in \Pi(I)$; (3) For each accumulation value $v \in \mathcal{A}$ that will be used in the following steps, it validates that v is indeed constructed from D by using the root element of $\mathcal{T}$; and (4) it verifies that I is indeed the correct intersection of all transactions in T^I by using $\mathcal{W}$ and $\mathcal{C}$.

The completeness verification is very similar to the correctness verification, except the first step by which the client verifies whether $|\mathcal{B}| \leq min_{sup}$.

The complexity of the verification of a single itemset I is $O(|I|)$. Given a set of itemsets F, the complexity of verifying its correctness/completeness is $O(Z)$, where $Z = \sum_{\forall I \in F} |I|$. This shows that the required complexity of the verification depends only on the outcome of frequent itemset mining, but not on the size of the involved dataset. This makes our verification approaches especially suitable for outsourcing of large datasets.

3) Discussion: The basic verification approach is expensive, due to the fact that the total number of (correctness and completeness) proofs is $2^m - 1$, where m is the number of unique items of D. This will introduce considerable amounts of overhead at both the client and the server sides.

C. Verification Optimization

In this section, we discuss how to reduce the number of proofs for verification to improve the performance of verification at both server and client sides.

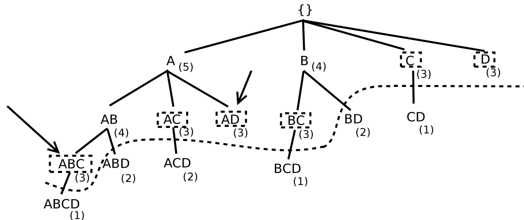


Fig. 3: An example of $LTST$ tree; nodes in dotted rectangles are positive border nodes ($min_{sup} = 3$)

Before we discuss our optimization strategy, we introduce the *lexicographic transaction subset tree* ($LTST$), which is adapted from [24]. Given a dataset D, assume there is a total lexicographic ordering of the items in the database. We say $i \leq j$ if item i occurs before item j in the ordering. We construct the *transaction subset tree* with the root being an empty set, while each internal node at level h of the tree contains a unique h-itemset *that exists in* D. There is an edge between a node N_i at level h and a node N_j at level $(h+1)$ if their corresponding itemsets $I_i = \{i_1, \dots, i_h\}$ and $I_j = \{i_1, \dots, i_{h+1}\}$ satisfy that $I_j - I_i = \{i_{h+1}\}$, where i_{h+1} is immediately after i_h in the ordering. Figure 3 shows an example of $LTST$. Note that the downward closure of the infrequentness and the upward closure of the frequentness are hold in $LTST$. We decide the lexicographic order of the items by their frequency. The items of higher frequency are assigned with smaller orders. For those items of the same frequency, we break the tie by assigning an ordering of these items. The lexicographic ordering is guaranteed to be a total ordering scheme.

Given two $LTST$ nodes N and N', we say N *consumes* N', denoted as $N' \preceq N$, if the itemset that N' correspond to is contained in the itemset that N correspond to. Next, we define the *(maximal) positive border* and *(minimal) negative border* of a $LTST$ tree.

Definition 4.1: **[Maximal Positive Border (MPB)]** Given a $LTST$ $\mathcal{L}$, its *positive border* (PB) consists of the nodes that corresponds to frequent itemsets but none of their children correspond frequent itemsets. Furthermore, we say a positive border node P is *maximal* if $\nexists$ a positive border node P' such that $P \preceq P'$.
For those nodes that correspond to frequent itemsets but have no children, we consider them as PB nodes too.

Definition 4.2: **[Minimal Negative Border (MNB)]** Given a $LTST$ $\mathcal{L}$, its *negative border* consists of the nodes that corresponds to infrequent itemsets while their parents correspond to frequent itemsets. Furthermore, we say a negative border node P is *minimal* if $\nexists$ a negative border node P' such that $P' \preceq P$.

Consider the $LTST$ $\mathcal{L}$ in Figure 3 as an example, assume $min_{sup} = 3$. The positive border consist of ABC, AC, AD, BC, C, and D, and MPB consists of ABC and AD. Unlike the problem of mining the frequent itemsets that tries to find the positive and negative borders, our problem is to verify whether the MPB and MNB discovered by the server is correct. The verification of MPB ensures the correctness of returned itemsets, while the verification of MNB verifies the completeness of the returned itemsets.

Next we describe an efficient algorithm to find MPB and MNB of the server's returned answer F^S. Given a set of frequent itemsets F^S that is returned by the server, assume each of them is ordered by the lexicographic order of items. We traverse the $LTST$ in a top-down fashion; for any node that corresponds to an itemset in F^S, we mark it as frequent. After we finish node marking, on each path, we pick the lowest node that is marked as frequent as a positive border node, and the highest node that is not marked as frequent as a negative border node. The MPB (MNB, resp.) nodes are collected as those nodes whose corresponding itemsets are not contained in (contain, resp.) any other positive border (negative border, resp.) nodes.

D. Proof Construction at Server Side

To prepare for correctness verification, the server constructs a cryptographic proof [21] for each MPB node. Similarly, to prepare for completeness verification, the server constructs a cryptographic proof for each MNB node. The MNB nodes should include all infrequent 1-itemsets. To further reduce the number of proofs, the server only needs to construct the proofs of those MNB nodes of length greater than one that only contain frequent 1-itemsets. After the proof construction is finished, the server sends the proofs with F^S to the client.

E. Verification at Client Side

Correctness verification. The correctness verification at the client side is straightforward. The client uses the server's proofs to verify that the itemset that each MPB node corresponds to is frequent by running the set intersection verification protocol. Though simple, sending proofs of only MPB nodes to the client raises two new issues. The first issue is that the client must ensure that the server has computed the proof of all MPB nodes of F^S; missing the proof of any MPB node may enable the server to escape from the verification. A naive method to verify the completeness of MPB nodes is that the client re-computes MPB from F^S, which may end up with high time cost. We design a more efficient method as following. Given F^S and the MPB nodes that are returned by the server, for each itemset $I \in F^S$, the client verifies whether there exists a MPB node whose corresponding itemset I' satisfy that $I \subseteq I'$. If it does not, the client concludes that the server misses the proof of at least one MPB node.

Another issue is that the client needs to ensure that all MPB nodes are honestly constructed from F^S, not from the (correct) mining result of D. Otherwise, the server may be able to escape from the verification. This can be achieved by verifying: (1) for each MPB node P, whether there exists an itemset $I' \in F^S$ such that $I' \subseteq I$, where I is the corresponding itemset of P; and (2) for each itemset $I \in F^S$, whether there exists a MPB node whose corresponding itemset I' satisfies that $I \subseteq I'$.

Completeness Verification. First, the client has to verify that the server does not miss the proof of any MNB node. To do this, the client can construct MNB from MPB, assuming that it has verified that the proof of MPB nodes is correct and complete.

Assume now the server has passed the verification of MNB nodes, next, the client uses the proof of MNB nodes to prove the completeness of returned frequent itemsets (i.e., each itemset that is not returned must be infrequent). Before we discuss how the client verifies the completeness, we categorize the possible missing frequent itemsets into four types, based on their relationships with the returned frequent itemsets F^S. In particular, consider a frequent itemset I that is not returned by the server, it should belong to one of the following four types:

- **Type-1 (non-overlap):** For each itemset $I' \in F^S$, $I \cap I' = \emptyset$.
- **Type-2 (subset):** There exists a frequent itemset $I' \in F^S$ such that $I \subseteq I'$.
- **Type-3 (superset):** There exists a frequent itemset $I' \in F^S$ such that $I' \subseteq I$.
- **Type-4 (overlap):** There is no frequent itemset $I' \in F^S$ such that $I \subseteq I'$ or $I' \subseteq I$. However, there exists a frequent itemset $I' \in F^S$ such that $I \cap I' \neq \emptyset$.

Next, we describe how to verify these four types of missing frequent itemsets respectively.

Verification of type-1 missing itemsets. Since type-1 itemsets do not overlap with any frequent itemset in F^S, any type-1 missing itemset must only contain frequent 1-itemsets that do not appear in F^S. Therefore, to verify whether there is any type-1 missing frequent itemset, the client checks whether all frequent 1-itemsets of D are included in F^S. If it does not, the client can conclude that there must exist at least one type-1 missing frequent itemset. The complexity of verification is $O(|F^S|)$, as its major effort is to collect unique items in F^S. Proof-based verification is not needed. The frequent 1-itemsets are collected when the client constructs the inverted index.

Verification of type-2 missing itemsets. This is equivalent to verifying whether all subsets of each itemset $I \in F^S$ also exist in F^S. To do this, the client verifies whether for each MPB node, all subsets of its corresponding itemset are included in F^S. In this case, as long as I is frequent (verified by correctness verification), this verification procedure does not need to use any proof.

Verification of type-3 & type-4 missing itemsets. If there is any type-3/type-4 frequent itemset missing, the MNBs of F^S include at least one node whose corresponding itemset is frequent. Therefore, the client verifies whether there exists any type-3/type-4 missing frequent itemsets by verifying the infrequentness of each MNB node via its proof (Section IV-B). If there exists any MNB node whose proof cannot show it is infrequent, there exists at least one type-3/type-4 missing itemset. The complexity of verification is $O(X)$, where $X = \sum_{\forall I \in MNB} |I|$.

F. Security Analysis

In this section, we will prove that although we reduce the number of proofs greatly, our verification optimization still provides the same security guarantee as our basic approach which requires correctness proofs for all itemsets in F^S and completeness proofs for all itemsets that do not appear in F^S, denoted as $\overline{F^S}$. Following our verification procedure, the client only checks proofs of MPB and MNB nodes.

In the aspect of correctness verification, as now the client only verifies the frequentness of all MPB nodes, our optimization eliminates the correctness proofs of all itemsets in $F^S - MPB$. Next, we will show that for each itemset in $F^S - MPB$, its proof can be consumed by the proof of at least one itemset in MPB. Note that for each itemset I in $F^S - MPB$, there must exists an MPB itemset I' such that $I \subseteq I'$. According to the upward closure of frequentness property of frequent itemsets, the frequentness of the MPB itemset I' ensures the frequentness of I.

Our completeness verification optimization only needs the proofs of MNB nodes of F^S. Similar to the proof above, we will show that for each itemset in $\overline{F^S} - MNB$, its proof can be consumed by the proof of at least one itemset in MNB. Here for each itemset $I' \in \overline{F^S} - MNB$, there must exists an MNB itemset I' such that $I' \subseteq I$. According to the downward closure of infrequentness property, the infrequentness of the MNB itemset I' ensures the infrequentness of I.

G. Robustness Analysis

Robustness against Type-1 server. As the deterministic approach does not modify the outsourced data, the prior background knowledge about the data distribution does not

change. Therefore, the type-1 server is not able to escape the verification.

Robustness against Type-2 server. As discussed in Section IV-E and IV-F, our optimized deterministic approach is able to catch incorrect and incomplete mining result with 100% guarantee. The power of the verification approach relies on the cryptographic proof. The server may try to cheat the client by claiming some infrequent itemsets as frequent, and constructing (fake) proofs for these itemsets from the modified dataset with these itemsets inserted into some transactions that they do not belong to. For example, consider the transaction dataset in Figure 2 (a). Assume $min_{sup} = 4$. If the server intends to cheat the client with the claim that $I = \{i_2, i_3\}$ (support = 3) is a frequent itemset, it may insert i_3 into transaction T_8, and return the proof as following: (1) $\mathcal{B}$ constructed from the four claimed transactions $\{T_1,\ T_4,\ T_7,\ T_8\}$, (2) $\mathcal{A}$ from the original Merkle tree $\mathcal{T}$, and (3) the subset witness $\mathcal{W}$ and subset completeness proof $\mathcal{C}$ constructed from T_8' with i_3 inserted. Such proof will pass the verification of co-efficients and accumulation values, but fail the subset correctness proof as it uses the original correct $acc(L_3)$ but incorrect co-efficients. Therefore, even though the server is aware of the authenticated data structure and proof construction procedure, it is not able to construct a correctness proof for an infrequent itemset and a completeness proof for a frequent itemset.

H. Complexity Analysis

Proof construction at server side. The total complexity of proof construction is $O(Ylog^3Y + m^{\epsilon}logm)$, where $Y = \sum_{I\in MPB\cup MNB} |I|$, m is the number of unique items of D, and $\epsilon \in (0, 1)$ is a user-specified constant that is used to specify the number of levels of the Merkle tree.

Verification at client side. The complexity of type-3 & type-4 missing frequent itemsets of completeness verification dominates the completeness verification. Therefore the total complexity of verification is $O(Y)$, where $Y = \sum_{\forall I\in MNB\cup MPB} |I|$. It is linear to the size of the returned mining result.

Upper bound of size of MPB and MNB nodes. The complexity of both proof construction and verification is linear to the size of MNB and MPB nodes. The number of MPB and MNB nodes, as well as the length of their corresponding itemsets, relies on the data distribution of the outsourced dataset and the support threshold min_{sup}. But we can estimate the theoretical upperbound of total number of MNB and MPB nodes, as well as the total length of their corresponding itemsets. We do this by estimating the upperbound of number of NB and PB nodes. Assume there are p PB nodes. As all of their children are NB nodes, the maximum total number of such NB nodes is $m + (m-1) + \cdots + (m-p+1) = mp - \frac{(p-1)(p-2)}{2}$. The other NB nodes are those infrequent nodes which are not children of any PB node. The maximum of the total number of such nodes is $\binom{m}{[\frac{m}{2}]} - p$ (i.e., these NB nodes and all PB nodes appear at the same level $[\frac{m}{2}]$. Therefore, the maximum of total number of PB and NB nodes c can be computed as:

$$c = mp - \frac{p(p-1)}{2} + \binom{m}{[\frac{m}{2}]}.$$

The c value reaches the maximum when $p = m$. Therefore,

$$|MPB \cup MNB| \leq \frac{m^2}{2} + \frac{m}{2} + \binom{m}{[\frac{m}{2}]}.$$

These MPB and MNB nodes cover all possible itemsets of length $[\frac{m}{2}]$, and $(\frac{m^2}{2}+\frac{m}{2})$ itemsets of length $[\frac{m}{2}]+1$. Therefore, the upperbound of total length of the itemsets that MPB and MNB nodes correspond to is

$$Y \leq ([\frac{m}{2}]+1)(\frac{m^2}{2} + \frac{m}{2}) + [\frac{m}{2}]\binom{m}{[\frac{m}{2}]}.$$

I. Update Operations

Update on min_{sup}. The update on min_{sup} does not change the authentication data structure. Therefore, no update is needed at the client side.

Update on D. The update on D requires the client to update the data structure *partially*. Specifically, for any update of a leaf node L in the Merkle tree, only the nodes on the path from the root to L are to be updated. If the client inserts a transaction $T = \{i_1, \dots, i_z\}$ to D, T is to be added to the inverted lists $L_{i_1}, \dots, L_{i_z}$ of the items in T. All the leaf nodes of the items in T and their ancestors in the Merkle tree are to be updated. After updating the Merkle tree, the client sends the updated structure to the server, together with the updated data. The proof construction at the server side is similar to Section IV-D. Specifically, the server constructs new cryptographic proofs for new MPB and MNB nodes. The client verifies the results against proofs in the same way.

V. Discussion

A. Various Outsourcing Scenarios

There two types of outsourcing paradigm: (1) the *infrastructure-as-a-service (IaaS)* paradigm in which the client outsources both data and the data mining software to the service provide (server). The server provides storage and hardware for the computation. A typical IaaS example is Amazon EC2 Web Service; (2) the *software-as-a-service (SaaS)* paradigm in which the client only outsources data to the server, while the server runs its own mining software on the outsourced data. We believe that our verification methods can be applied to both paradigms. This is because that frequent itemset mining is deterministic in the sense that the output is fixed for a given dataset and min_{sup}, no matter how the algorithm is implemented.

B. Extension to Other Data Mining Algorithms

In general, our strategy of constructing cryptographic proofs for verification (for deterministic guarantee) and artificial verification objects (for probabilistic guarantee) can be applied to most data mining algorithms. The challenge is that the design of proofs and verification objects has to be customized for different data mining algorithms. Below we discuss briefly how to extend our approaches to association rule mining, which is related to frequent itemset mining. The association rule mining aims to find association rules that meet the pre-defined *support* and *confidence* threshold value min_{sup} and min_{conf}. First, for the probabilistic approach, we can build *evidence association*

Dataset	# of trans.	# of items	Avg. trans. length	min_{sup}	# of freq. itemsets
S_1	10^3	49	10	250	36
S_2	10^4	49	10	250	3854
S_3	10^5	49	10	250	149744
S_4	10^6	49	10	250	3074610
R_1	88162	16470	124	50 10	16778 155111
R_2	500	100	2.4	5	97
$NCDC$	500	365	332.9	450	559368361

TABLE I: Details of Datasets

rules $EF_i \rightarrow EF_j$ by constructing two *evidence frequent itemsets* EF_i and EF_j, requiring that $EF_i \subset EF_j$ and $\frac{sup_{D'}(EF_j)}{sup_{D'}(EF_i)} \geq min_{conf}$. EF_i and EF_j can be constructed by using our *MiniGraph* approach. Second, for the deterministic verification approach, the server returns the proofs of all frequent itemsets and MNB itemsets. The client checks if the server returns all the frequent itemsets from the proof. If the server does, for any pair of itemsets I_i and I_j such that $I_i \subset I_j$, the client calculates the confidence of $I_i \rightarrow I_j$ to check whether it is indeed an association rule.

VI. Experiments

We ran a battery of experiments to evaluate the performance of our probabilistic and deterministic approaches. We also have a comparison of our two approaches.

A. Setup

Hardware. We run our experiment on a Intel machine with 2.4GHz CPU, 4GB memory, running Mac OS X 10.7.5.

Datasets. We use the IBM generator to generate four synthetic datasets S_1, S_2, S_3, and S_4 of various sizes. We also use two real-world datasets named *Retail* dataset and *NCDC* dataset[1]. The *Retail* dataset is available at the Frequent Itemset Mining Dataset Repository[2]. The *Retail* dataset R_1 contains 88162 transactions and 16470 items. We also construct a small dataset R_2 from the *Retail* dataset that contains 500 transactions and 100 items. The *NCDC* dataset comes from National Climatic Data Center of U.S. Department of Commerce. Table I shows the details of the datasets and our mining setup. Among these datasets, the *NCDC* dataset is a *dense* dataset, in which most of the transactions are of similar length, and contain $> 75\%$ of items; and the R_1 dataset is a *sparse* dataset in which the transactions are of skewed length distribution. Due to

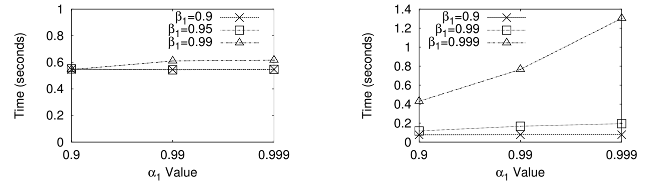
Simulation of malicious actions. We set the error ratio $p = 1\%, 2\%, 5\%, 10\%$, and 20%. For the simulation of incomplete result, we randomly pick p percent of frequent itemsets from the mining result and remove these picked itemsets. For the simulation of incorrect result, we randomly generate p percent of infrequent itemsets and insert them into the result.

Implementation. We implemented a prototype of our probabilistic approach in Java, and the deterministic approach in C++. We use the PBC library[3] to implement bilinear pairing, and SHA256 for the implementation of the hash function in Merkle tree construction. The accumulation values in Merkle tree are computed by using the element

[1]http://lwf.ncdc.noaa.gov/oa/climate/rcsg/datasets.html

[2]Frequent Itemset Mining Dataset Repository: http://fimi.ua.ac.be/data/.

[3]PBC library: http://crypto.stanford.edu/pbc/manual/.



(a) NCDC dataset (b) R1 dataset

Fig. 4: Time Performance of EF Construction

power function in the PBC library. The coefficients $\mathcal{W}$ in the proof are computed using the NTL library[4].

B. Probabilistic Approach

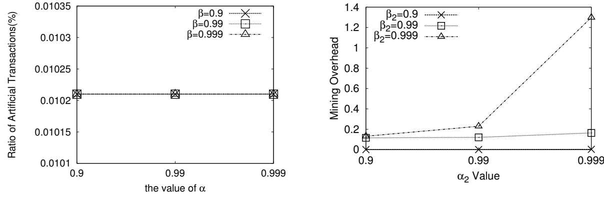
We measure the time performance and amount of noise brought to the original dataset by our probabilistic approach. We tried various settings for $\alpha_2, \alpha_1, \beta_2$ and β_1 values. In particular, we use $\alpha_2, \alpha_1 = 0.9, 0.99, 0.999$, and $\beta_2, \beta_1 = 0.9, 0.99, 0.999$. On *R1* dataset, we set the min_{sup} to be 10.

1) Robustness: We measure the robustness of our probabilistic approach by studying the probability that the incorrect/incomplete frequent itemsets can be caught by using artificial EIs/EFs. We use the *R1* dataset and vary α_1 and α_2 values to control the amount of mistakes that the server can make on the mining result. For each α_1 (α_2, resp.) value, we simulate the incomplete (incorrect, resp.) mining result. Then with various β_1 and β_2 values, we construct artificial tuples to satisfy (α_1, β_1)-completeness and (α_2, β_2)-correctness. Detection of any missing EF or the presence of any EIs will be recorded as a successful trial of catching the server. We repeat 1,000 times and record the percentage of trials (as *detection probability*) that the server is caught, with $\alpha_1, \alpha_2 \in [0.7, 0.9]$ and $\beta_1, \beta_2 \in [0.7, 0.9]$. It shows that the detection probability for the completeness and correctness verification is always higher than β_1 and β_2 respectively. This proves the robustness of our probabilistic approach. The results are omitted due to limited space.

2) Completeness Verification:

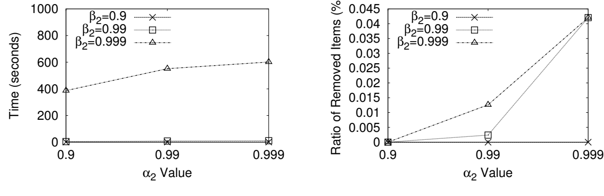
Time performance. We measure the time performance of EF construction with various α_1 and β_1 values. Figure 4 shows the result of using *NCDC* and *R1* datasets. For both datasets, the time of constructing EFs grows with the increase of α_1 when $\beta_1 = 0.99$ and 0.999. This is not surprising as higher completeness probability guarantee requires more EFs for verification. However, when $\beta_1 = 0.9$, the EF construction time decreases when α_1 increases from 0.9 to 0.99. This is because the size of D_h when $\alpha_1 = 0.99$ is smaller than the size of D_h when $\alpha_1 = 0.9$. Since the construction time relies on both $|EF|$ and $|D_h|$, the construction time decreases when α_1 increases. Another interesting observation is that the construction time on *NCDC* dataset grows faster than *R1* dataset. This is because the EF construction algorithm, which is driven by transaction length, will pick a large portion of transactions in *NCDC* dataset where most of transactions are of similar length but a small portion of

[4]NTL library: http://www.shoup.net/ntl/doc/tour-intro.html.



(a) Artificial Transactions Ratio (b) Mining Overhead

Fig. 5: Ratio of Artificial Transactions and Mining Overhead (R1 dataset)



(a) Time Performance (b) Ratio of Removed Items

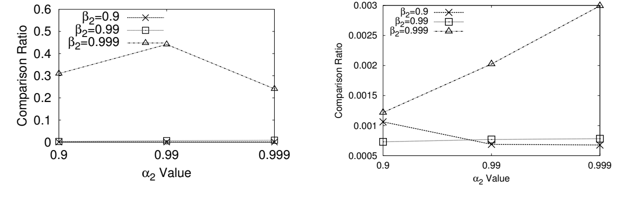
Fig. 6: EI construction (*NCDC* dataset)

transactions in *R1* dataset where transaction lengths are of skewed distribution.

Amounts of artificial transactions. We measured the amount of inserted artificial transactions compared with the size of the database. In particular, let t be the number of artificial transactions to be inserted, we measured the ratio $r = \frac{t}{n}$, where n is the number of real transactions in D. As shown in Figure 5 (a), for *Retail* dataset, the inserted artificial transactions only take a small portion of the original database (no more than 0.01025%). On the other hand, we observed that the ratio of artificial transactions is not increasing with the α and β values. That is because the number of artificial tuples is bounded to the min_{sup}, i.e., the number of artificial transactions is at most $min_{sup} - 1$. We also measured the mining overhead introduced by artificial transactions. Figure 5 (b) shows the result. It is expected that the mining overhead is negligible.

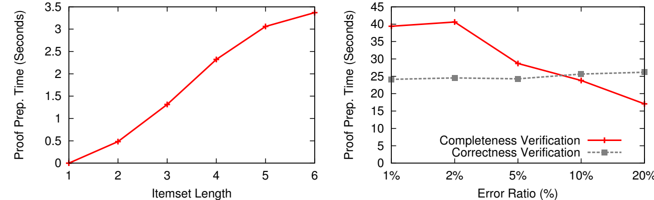
3) Correctness Verification: **Time Performance.** We measure the time for correctness verification preparation that satisfy (α_2, β_2)-correctness on $NCDC$ dataset. The result is shown in Figure 6 (a). It is not surprising that it needs more time to construct EIs for higher α_2 and β_2 values. Furthermore, when $\beta_2 = 0.9$ and 0.99, EI construction is very fast (no more than 1 second). This is because for these cases, all EIs are real infrequent itemsets; there is no need to remove any item. However, when β_2 grows to 0.999, the preparation time jumps to 400 - 600 seconds, since now the algorithm needs to find more itemset candidates from deeper level of the lattice to be EIs as well as the items to be removed to make EIs infrequent. We also measure the EI construction time of $R1$ dataset. It does not increase much when β_2 increases from 0.9 to 0.999, since all EIs are real infrequent items.

Amounts of removed items. We measure the amount of item instances that are removed by EI construction. In particular, let d be the number of item instances to be removed, we measure the ratio $r = \frac{d}{|D|}$. The result of $NCDC$ dataset is shown in Figure 6 (b). It can be seen that the number of item instances to be removed is a negligible portion (no more than 0.045%) of $NCDC$ dataset. There



(a) EI construction time VS. mining time (b) EF construction time VS. mining time

Fig. 7: Mining V.S. Verification Preparation ($NCDC$ dataset)



(a) Time of constructing one proof (b) Total proof preparation time

Fig. 8: Proof Construction Time ($D_{500\times100}$)

is no item that is removed from $R1$ dataset, as it has a large number of infrequent 1-itemsets, which provides sufficient number of EI candidates. This shows that we can achieve high correctness guarantee to catch small errors by slight change of the dataset.

4) Outsourcing versus mining locally.: We also compared the time of executing frequent itemset mining locally with the time of verification preparation by measuring the ratio t_{VP}/t_{DM}, where t_{VP} and t_{DM} are the time of verification preparation and frequent itemset mining. Figure 7 shows the result of $NCDC$ dataset. It can be seen that our verification preparation is always cheaper than mining locally. This convinces the client to outsource her data mining task to the server and prepare for verification locally with affordable overhead.

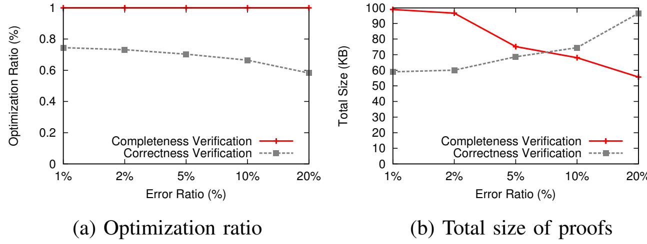
C. Deterministic Approach

In this section, we measure the performance of proof construction at the server side and verification at the client side and explored various factors that impact the verification performance of our deterministic approach, including various error ratio, frequent itemsets of different lengths, and different database sizes. We set the support threshold on *R1* dataset to be 50.

1) Proof Construction at the Server Side: In this section, we show the experiments of proof preparation at the server side.

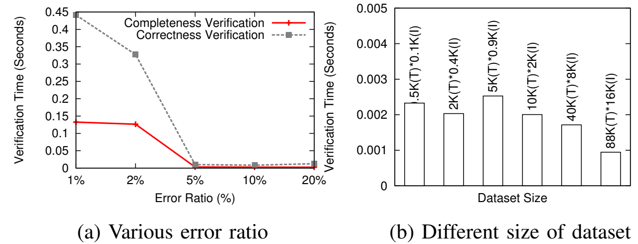
Time performance. First, we measure the time of preparing one single proof of (in)frequent itemsets of various lengths. We show the preparation time for itemsets of size up to six because in our main experiment, the size of itemsets to be verified is no more than six. As shown in Figure 8 (a), the proof preparation time increases with the length of frequent itemset. This is consistent with our theoretical analysis (Section IV-H) that the complexity of proof construction is dependent on the size of mining result, not on the input dataset.

Second, we measure the total time of preparing all the proofs for verification. Figure 8 (b) shows the total



(a) Optimization ratio (b) Total size of proofs

Fig. 9: Proof size ($D_{500\times100}$)



(a) Various error ratio ($D_{500\times100}$) (b) Different size of dataset (error ratio = 20%)

Fig. 10: Verification Time at the Client Side

preparation time for various error ratios. We consider the same error ratios $p = 1\%, 2\%, 5\%, 10\%$, and 20% that we used for simulation of malicious actions. First, we observe that for correctness verification the proof preparation time increases slowly with the increase of error ratio. It is obvious that adding infrequent itemsets to the mining result leads to more MPB nodes. Indeed, the number of MPB nodes change much. For example, when p changes from 10% to 20%, the number of MPB nodes changes from 32 to 40. However, the total proof construction time does not increase much, since the proofs of new MPB nodes, which indeed correspond to infrequent itemsets, can be constructed very fast, due to the reason that the polynomial P_j [21] used in their subset witness and set completeness witness is of low degree. Second, the result shows that the proof preparation time for completeness verification decreases when the error ratio increases. We studied how MNB nodes change when the error ratio grows. Our results show that the MNB nodes move to the higher levels of the $LTST$ tree when more frequent itemsets are missing. This is because now those missing frequent itemsets were changed to be NB nodes, while their ancestor nodes in the $LTST$ tree become MPB nodes. As MNB nodes are constructed from the children and sibling of MPB nodes, there are fewer MNB nodes especially due to the fact that higher $LTST$ nodes have fewer children. For instance, when the error ratio changes from 2% to 5%, the number of completeness proofs changes from 33 to 27.

Proof size. First, we measure the optimization ratio of proofs for the mining results of various error ratios. We define the optimization ratio as $1 - p_1/p_1$, where p_1 is the number of proofs by our optimization, and p_2 is the number of proofs by the basic approach. Intuitively, the closer the optimization to 1, the better. Figure 9 (a) displays the optimization ratio. We observe that the optimization ratio of the number of completeness proofs is very close to 1. This is because the basic verification approach needs to construct $(2^{|\mathcal{I}|} - |F^S|)$ proofs (for this experiment $|\mathcal{I}| = 100$ and $|F^S| = 97$) for completeness verification, our optimized approach only needs no more than 34 completeness proofs. This proves that our optimization can reduce the number of proofs dramatically. We also observe that the optimization ratio of the number of correctness proofs deceases when the error ratio increases. The reason is that adding infrequent itemsets into the result leads to more MPB nodes and thus the number of proofs, which decreases the optimization ratio. Nevertheless, the optimization ratio of the number of correctness proofs is high; it is at least 0.6 even when the error ratio is 20%.

Second, we measure the total size of the proofs for the mining result with various error ratio. Figure 9 (b) shows the result. It is not surprising that the trend of total size of the proofs is consistent with the trend of number of proofs (Figure 9 (a)). In particular, the total size of completeness proofs decreases dramatically when the error ratio grows. The reason is similar to our analysis of the number of proofs: as MNB nodes move to the higher levels of the $LTST$ tree, both the number of MNB nodes and the length of their corresponding itemsets decrease. This lead to the decrease of the total size of the completeness proofs. Regarding the correctness proofs, since the number of MPB nodes and the number of correctness proofs increase, the total size of correctness proofs increases.

We also measure the total size of the proofs, with MPB and MNB nodes of various lengths. It is not surprising that the size of proofs grows with itemsets of larger size, as the space complexity is dependent on the size of output frequent itemsets. We omit the results due to limited space.

2) Verification Time at the Client Side: First, we measure the verification time at the client side for mining result with various error ratios. From the result shown in Figure 10 (a), first, we observe that both completeness and correctness verification are very fast. Second, we observe that the completeness verification time decreases dramatically when the error ratio increases from 2% to 5%, then keeps stable afterwards. We analyzed the reason, and it turned out that when the ratio equals to 2% (1% as well), the incomplete itemsets were caught by checking against the proofs, while for the error ratio changes to 5% and more, the incomplete itemsets were caught by missing at least one proper subset or one frequent 1-itemset, which is much faster than using proofs. Third, we observe that the correctness verification time drops sharply when the error ratio changes from 1% to 5%. The reason is that higher error ratio leads to larger chance that the client catches an infrequent itemset earlier. When the error ratio increases from 5% to 20%, the verification time is stable because now the client can catch the infrequent itemset by the first trial.

To measure the scalability of our verification approach, we measure the verification time on datasets of different sizes. The result is shown in Figure 10 (b). We observe that our verification scheme is very fast even for very large datasets. For example, for the dataset of 88162 transactions, it only requires 0.001045 seconds for verification.

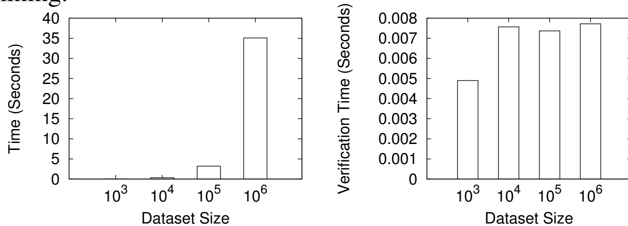
3) Client VS. Server: We compared the time performance at both the client and the server sides for various support threshold values. We vary the support threshold values

so that the number of frequent itemsets is approximately 1%, 5%, and 10% of the number of transactions. Table II shows the comparison result. First, it is not surprising that the verification time at the client side increases with the growth of the number of frequent itemsets. The same pattern also holds for the server side. Second, in all of our testings, the total overhead at the client side is much smaller than that at the server side. Furthermore, the verification procedure at the client side is much faster than the mining at the server side. This proves that the client can outsource the mining task to the server, while verifying the result integrity with the cost that is cheaper than mining locally.

min_{sup}	# of Freq.	Client side	Server side	
	Itemsets	Verify	Proof prep.	mining
402	10	0.000164	24.72	0.03707
203	50	0.001358	266.985	0.08984
157	99	0.00332	572.591	0.1355

TABLE II: Client VS. Server w.r.t Time Performance (S_1 dataset)

4) Scalability: We measured the time performance on the datasets of various sizes. Figure 11 (a) shows that the time of constructing one single proof increases with the data size. However, the proof can be constructed fast; it only needs 35 seconds even for a dataset of 10^6 records. Figure 11 (b) shows that the verification time does not change much with the data size, since: (1) the verification complexity is decided by the number of MII and MFI itemsets, and (2) the verification procedure always catches the first incomplete itemset by type-1/2 checking for most of the cases. Similar observations hold on the *Retail* dataset (88162 transactions); it only requires 0.001045 seconds for verification. This convinces us that our approach can be used for efficient verification of outsourced frequent itemset mining.



(a) Construction time of one proof (itemset length = 3)

(b) Client verification time

Fig. 11: Scalability (error ratio=1%)

D. Probabilistic VS. Deterministic Approaches

We ran experiments to compare the performance of our probabilistic and deterministic approaches. Table III shows the comparison result on S_3 dataset of various settings. We pick the error ratios of 1%, and vary the probabilistic guarantee threshold from 90% to 100% (probability = 100% corresponds to our deterministic approach). Table III shows the details of the comparison result. In general, the deterministic approach brings higher overhead at the server side than the probabilistic approach. However, this is the sacrifice that we have to pay for higher result integrity guarantee. We also observe that in some cases (marked as N/A in Table III), the probabilistic approach fails as it cannot provide required probabilistic correctness guarantee due to the data distribution. The deterministic approach does not have such limit.

Error ratio	Integrity Prob.	Type	Client	Server	
			Verify	Proof prep.	Mining
1%	90%	R	N/A	0	0.042
		M	1.433	0	1024.53
1%	95%	R	N/A	0	0.042
		M	0.945	0	1204.59
1%	99%	R	N/A	0	0.042
		M	0.689	0	1498.67
1%	100%	R	0.000628	1660.12	0.5707
		M	0.4123	2785.6	0.5707

TABLE III: Time performance: deterministic V.S. probabilistic approaches (S_3 dataset; R: correctness, M: completeness)

E. Comparison with Existing Work

Among all the related work, [14] and [22] are the closest to ours. It has been proven that the evidence patterns constructed by the encoding method in [14] can be identified even by an attacker without priori knowledge of the data [11]. We argue that our probabilistic verification approach is robust against the attack in [11]. Besides, our probabilistic approach is more efficient. In [14], it shows that it may take 2 seconds to generate one evidence pattern, while our method only takes 600 seconds to generate 6900 evidence itemsets (i.e., 0.09 second per pattern).

Goodrich el al. [22] propose an efficient cryptographic approach to verify the result integrity of web-content searching by using the same set intersection verification protocol as ours. It shows that the time spent on the server to construct the proof for a query that involves two terms is between 0.35 to 0.8 seconds. Our deterministic approach requires 0.5 seconds to construct the proof for an itemset of length 2 at average, which is comparable to the performance of [22].

VII. Related Work

The problem of verifiable computation was tackled previously by using interactive proofs [7], probabilistically checkable proofs [2], and non-interactive verifiable computing [5]. This body of theory is impractical [13], due to the complexity of the algorithms and difficulty to use general-purpose cryptographic techniques in practical data mining problems. To reduce the complexity, Goodrich et al. [21] design efficient cryptographic approaches to verify the correctness of keyword search by using Bilinear pairings and Merkle hash trees. A large body of work design authenticated data structures (e.g., [21], [12]) that provide a cryptographic proof of the answer to a query. These work focus on query evaluation and set operations.

In the last decade, intensive efforts have been put on the security issues of the database-as-a-service (DaS) paradigm (e.g., [8], [9]). The focus is the correctness of SQL query evaluation. Only until recently some attention was paid to the security issues of the data-mining-as-a-service ($DMaS$) paradigm [11], [6]. However, most of these work only focus on how to encrypt the data to protect data confidentiality and pattern privacy, while we focus on integrity verification of mining result.

There is surprisingly very little research [14], [10] on result correctness verification in the $DMaS$ paradigm. Among these work, only [14] focused on integrity verification methods for frequent itemset mining. Its basic

idea is to insert some *fake items* that do not exist in the original dataset into the outsourced data; these fake items will construct a set of fake (in)frequent itemsets. The correctness of mining results is verified against the fake (in)frequent itemsets. Though effective, this method assumes that the server has no background knowledge of the items in the outsourced datasets, and thus it has equal probability to cheat on the fake and true itemsets. We argue that using fake items cannot catch our type-1 malicious server that may have some background knowledge of the original data and be able to escape the verification by distinguishing between real and fake items. We aim at designing effective verification approaches that can catch the incomplete/incorrect frequent itemsets from such a malicious server.

VIII. Conclusion

In this paper, we present two integrity verification approaches for outsourced frequent itemset mining. The probabilistic verification approach constructs evidence (in)frequent itemsets. In particular, we remove a small set of items from the original dataset and insert a small set of artificial transactions into the dataset to construct evidence (in)frequent itemsets. The deterministic approaches requires the server to construct cryptographic proofs of the mining result. The correctness and completeness are measured against the proofs with 100% certainty. Our experiments show the efficiency and effectiveness of our approaches.

An interesting direction to explore is to extend the model to allow the client to specify her verification needs in terms of budget (possibly in monetary format) besides precision and recall threshold.

IX. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. #1350324.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[2] Laszlo Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *STOC*, pages 21–32, 1991.

[3] Ran Canetti, Ben Riva, and Guy N. Rothblum. Verifiable computation with two or more clouds. In *Workshop on Cryptography and Security in Clouds*, 2011.

[4] Kun-Ta Chuang, Jiun-Long Huang, and Ming-Syan Chen. Power-law relationship and self-similarity in the itemset support distribution: analysis and applications. *The VLDB Journal*, 17:1121–1141, August 2008.

[5] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482, 2010.

[6] Fosca Giannotti, Laks V. S. Lakshmanan, Anna Monreale, Dino Pedreschi, and Wendy Hui Wang. Privacy-preserving data mining from outsourced databases. In *Computers, Privacy and Data Protection*, pages 411–426. 2011.

[7] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal of Computing*, 18:186–208, February 1989.

[8] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

[9] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, pages 121–132, 2006.

[10] Ruilin Liu, Hui Wang, Anna Monreale, Dino Pedreschi, Fosca Giannotti, and WengeGuo. Audio: An integrity auditing framework of outlier-mining-as-a-service systems. In *ECML/PKDD*, 2012.

[11] Ian Molloy, Ninghui Li, and Tiancheng Li. On the (in)security and (im)practicality of outsourcing precise association rule mining. In *ICDM*, pages 872–877, 2009.

[12] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *CCS*, pages 437–448, 2008.

[13] Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, 2011.

[14] W. K. Wong, David W. Cheung, Ben Kao, Edward Hung, and Nikos Mamoulis. An audit environment for outsourcing of frequent itemset mining. In *PVLDB*, volume 2, pages 1162–1172, 2009.

[15] Feida Zhu, Xifeng Yan, Jiawei Han, Philip S. Yu, and Hong Cheng. Mining colossal frequent patterns by core pattern fusion. In *ICDE*, 2007.

[16] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, 2011.

[17] Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In *CCS*, 2011.

[18] Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *CCS*, 2012.

[19] Menezes, Alfred and Vanstone, Scott and Okamoto, Tatsuaki. Reducing elliptic curve logarithms to logarithms in a finite field. In *STOC*, 1991.

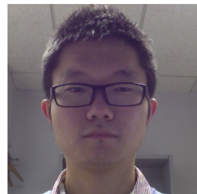
[20] R.C.Merkle. Protocols for public key cryptosystems. In *Symposium on Security and Privacy*, 1980.

[21] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, 2011.

[22] Michael T. Goodrich, Charalampos Papamanthou, Duy Nguyen, Roberto Tamassia, Cristina Videira Lopes, Olga Ohrimenko and Nikos Triandopoulos Efficient Verification of Web-Content Searching Through Authenticated Web Crawlers In *PVLDB*, volume 5, pages 920–931, 2012

[23] Bryan Parno, Mariana Raykova, and Vinod Vaikuntanathan. How to delegate and verify in public: verifiable computation from attribute-based encryption. In *TCC*, 2012.

[24] D. Burdick, M. Calimlim, J. Gehrke, MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2001.



Boxiang Dong received the BE degree from Dalian University of Technology, Dalian, China, in 2011. He is a Ph.D student in the Computer Science Department, Stevens Institute of Technology, New Jersey. His research interests include data mining, anomaly detection, data security and privacy.

Ruilin Liu is a PhD candidate at the Computer Science Department of Stevens Institute of Technology. He received his Bachelor degree from Southwest University of Science and Technology (China), in 2007. His research interests include anomaly detection, data mining integrity verification, privacy-preserving data mining.

Dr. Wendy Hui Wang is an assistant professor in the Computer Science Department, Stevens Institute of Technology, New Jersey. She received her PhD degree in computer science from University of British Columbia, Vancouver, Canada. Her research interests include data management, data mining, database security, and data privacy. She is a member of the editorial boards of Journal of Information Technology and Architecture and Journal of Information Systems.