Speeding Up Secure Computations via Embedded Caching

K. Zhai[*] W. K. Ng[†] A. R. Herianto[‡] S. Han[§]

Abstract

Most existing work on Privacy-Preserving Data Mining (PPDM) focus on enabling conventional data mining algorithms with the ability to run in a secure manner in a multi-party setting. Although various algorithms in data mining have been enhanced to incorporate secure mechanisms for data privacy preservation, their computation performance is far too high to allow them to be practically useful. This is especially true for those algorithms that make use of common cryptosystems. In this paper, we address the efficiency issue of PPDM algorithms by proposing to cache result data that are used more than once by secure computations. For this to be possible, we carefully examine the micro steps of secure computations to identify the repetitive or iterative portions and reduce the overall computational cost by caching intermediate results/data. We have applied this to decision tree induction, association rule mining and k-means clustering that make use of secure building blocks such as secure multi-party sum, secure matrix multiplication, and secure inverse of matrix sum. We show empirically that the computational costs of secure computations can be reduced without affecting the quality of the data mining result in general. Our experiments show that the caching technique is generalizable to common data mining algorithms and the efficiency of PPDM algorithms can be greatly improved without compromising data privacy.

1 Introduction

PPDM enables conventional data mining algorithms to be securely performed in a distributed, multi-party environment with sensitive data. To date, a host of data mining algorithms have been retrofitted to incorporate privacy preservation. Generally, there are two approaches for PPDM algorithm: randomization [2] and secure multi-party computations [9]. In this paper, we focus on PPDM algorithms that make use of secure computations as the outcome of this category of algorithms does not compromise the correctness of the algorithm to a large extent. Approaches based on secure multi-party computation are too costly for practical applications. In the Workshop on Practical Privacy-Preserving Data Mining (P3DM'08) [10] held in conjunction with SIAM SDM'08, it has been acknowledged that improving the efficiency of PPDM is a challenge. Our focus in this paper is to improve the computational/communication efficiency of PPDM algorithms that make use of secure multi-party computations.

We observe that in general, many data mining algorithms are iterative in nature and execute certain data operations repetitively. When such data are distributed among multiple parties and the data are sensitive, the parties invoke secure computations that involve encryptions/decryptions and network communications. Hence, the cost of secure computations is higher than the non-secure version, and it renders PPDM algorithms less practical for general usage.

We illustrate with two examples. In the Privacy-Preserving k-means Clustering (PPKC) algorithm that was proposed by Jagannathan and Wright [8], one needs to compute the distance between each data point and the cluster center in each iteration. This incurrs high computation/communication costs. As this distance is computed using the Secure Scalar Product (SSP) protocol, encryption of the input data can be performed once and reused many times subsequently. To achieve this, we could cache the essential intermediate results to increase the efficiency. From the experiments performed in this paper, we have indeed shown that the running time can be dramatically reduced. Moreover, the modified secure protocol that incorporates caching is much more scalable in terms of efficiency.

Similar inefficiency is also observed in the Privacy-Preserving Association Rule Mining (PPARM) algorithm by Vaidya and Clifton [13]. At each iteration of computing the frequency count of every itemset, the same data vectors that are distributed among different parties are used. Each time such vectors need to be used by the parties, several SSP operations are invoked. As pointed out by Subramaniam *et al.* [12], high computational overheads

[*]School of Computer Engineering, Nanyang Technological University, Singapore, zhai0005@ntu.edu.sg

[†]School of Computer Engineering, Nanyang Technological University, Singapore, wkn@acm.org

[‡]School of Computer Engineering, Nanyang Technological University, Singapore, andr0027@ntu.edu.sg

[§]School of Computer Engineering, Nanyang Technological University, Singapore, hans0004@ntu.edu.sg

have become a major performance bottleneck of SSP protocols. These repeated computational overheads reduce the efficiency of PPARM. We performed a careful micro-inspection of the algorithm and associated protocols and found ways to embed caching and avoid redundant computations; thus, improving the overall efficiency.

Based on the above observations, we want to improve the efficiency of the PPDM algorithms by cutting down the computational and communication overheads incurred by secure computations. To accomplish this objective, we identify a set of secure computations and examine their micro steps to identify the possibility of embedding caching. Some of these micro steps involve cryptographic operations on data. In this paper, we perform a micro-inspection of the Secure Sum, SSP and Secure Matrix Multiplication (SMM) protocols. These secure computations are used in high-level PPDM algorithms such as association rule mining, decision tree, and k-means clustering.

We believe our work contributes to an important aspect of privacy-preserving data mining—computational/communication efficiency—that has largely been ignored by the PPDM community. The work is part of our larger efforts to develop a practically feasible PPDM system that supports multiple distributed parties, much like a secure, privacy-preserving version of Weka that can be plugged into database systems. Our work generalizes various computational properties used in secure computations, such as the downward closure property and homomorphic encryption, so that computation results are cachable. We theoretically and experimentally show that the caching approach is generalizable to other data mining algorithms.

This paper is organized as follows: The next section summarizes a number of existing PPDM algorithms that make use of secure computations. Section 3 presents details on several existing secure computations that include observations of properties of computations and whether these computations support caching. We also explain how the caching technique can be applied to algorithms elaborated in Section 2. In Section 4, we analyze the efficiency improvements of caching and make comparisons with conventional data mining algorithms without caching. We conclude the paper in the final section.

2 Related Work

In this section, we review some secure building blocks and conventional data mining algorithms that are widely used. To date, research in privacy-preserving data mining has produced a host of secure computation protocols such as secure multi-party sum [3], secure comparison [15], secure scalar product [5] for data mining algorithms such as decision tree [14], association rule mining [13], k-means clustering [8], and machine learning algorithms such as fisher discriminant analysis [6], and singular values decomposition [7]. In the following sections, we describe some specific data mining algorithms. In the next section, we show how the caching idea can be applied to secure building blocks to increase their efficiency.

2.1 Secure Scalar Product The SSP protocol is a protocol to compute the scalar product of two vectors from two parties without any private data disclosure. It is one of the most important secure computations protocols that are frequently used in many PPDM algorithms [7, 6, 8, 13]. One of the most well known SSP protocols is the one proposed by Goethals *et al.* [5] which makes use of a cryptosystem that supports homomorphic addition property: $E(x+y) = E(x)E(y)$. This property gives rise to the following:

$$E(\mathbf{x}\cdot\mathbf{y}) = E\left(\sum_{i=1}^{n}(x_i y_i)\right) = \prod_{i=1}^{n} E(x_i y_i) = \prod_{i=1}^{n} E(x_i)^{y_i}$$

The algorithm goes as follows: Party A holding vector $\mathbf{x}$ uses the public key to encrypt all elements and sends them to Party B. Party B computes $\prod_{i=1}^{n}(E(x_i)^{y_i})$ and then sends them back to Party A. Party A decrypts and obtains the scalar product value.

2.2 Secure Multi-party Sum Yao [15] first introduced the general concept of Secure Multi-party Computation (SMC). The Secure Multi-party Sum (SMS) was demonstrated by Clifton *et al.* [3] to allow multiple parties to compute the sum of their private shares together without disclosing the information to any other party. The main idea of the algorithm is that the first party uses one random number to hide its private number, and then send the sum of private number and random number to the second party. The second party adds its private number to the sum received from the first party and then sends the new sum to third party. After one round, the final party sends the sum to the first party. The first party subtracts the random number and obtains the total sum.

2.3 Privacy-Preserving Cooperative Linear System of Equations The Privacy-Preserving Cooperative Linear System of Equations Protocol (PP-CLSE) was proposed by Du and Atallah [4] to find the solution of linear systems of equations collaboratively. This protocol is applicable to linear regression;

thus, further applicable to finding classification rules with disclosure of each party's private data. The dataset may be partitioned among parties vertically, horizontally, arbitrarily or even overlapping. This protocol works securely and accurately regardless of the partition scheme adopted by the parties.

As an example, suppose Alice holds an $n \times n$ matrix M_1 and an n dimension vector b_1. Bob holds an $n \times n$ matrix M_2 and an n dimension vector b_2. The protocol is based on the solution to the linear system of equations $P(M_1+M_2)QQ^{-1}\boldsymbol{x} = P(b_1+b_2)$, which is equivalent to the solution of the system of equations $(M_1 + M_2)\boldsymbol{x} = b_1 + b_2$. In order to solve the system of equations, Bob generates two random matrices P and Q and compute the matrix $M^{'} = P(M_1 + M_2)Q$ and $b^{'} = P(b_1 + b_2)$ without disclosing any private information. They both obtain the solution $\boldsymbol{x}$ consequently.

2.4 Privacy-Preserving Association Rule Mining Vaidya and Clifton [13] proposed an algorithm for PPARM on vertically partitioned data. Attribute values in this algorithm are binary where 0 represents the absence and 1 represents the presence of the attribute value. This algorithm is based on the conventional *Apriori* algorithm for multi-party privacy-preserving Apriori. The cornerstone of PPARM is to compute the frequency of itemsets. As an itemset could be partially owned by two parties, a SSP operation is required.

2.5 Privacy-Preserving Decision Tree Induction Vaidya and Clifton [14] proposed a Privacy-Preserving ID3 (PPID3) algorithm for constructing a decision tree classifier on vertically partitioned data. The core operation of ID3, as well as the PPID3 algorithm, is to compute the information gain for a specific set of attributes in order to determine the best splitting attribute. A SSP computation is also required to evaluate the entropy on certain attributes to further build the decision tree.

2.6 Privacy-Preserving k-Means Clustering Jagannathan and Wright [8] proposed an algorithm for distributed PPKC on arbitrarily partitioned data that retrofits the conventional k-means algorithm for a two-party scenario. The basic idea is to constantly update every cluster center until a stable state is achieved. The critical operation is to compute the Euclidean distance between a data point and the cluster center; this requires invocation of SSP as the data point is partially owned by two parties. Furthermore, the termination state also needs a series of SSP computations to be performed.

2.7 Secure Matrix Multiplication & Inverse of Matrix Sum Computation Han and Ng [6] addressed the secure multiplication of two matrices. Party A and B each hold a private $m \times N$ matrix A and $N \times n$ matrix B respectively. Using of SSP protocol in Section 2.1, the matrix multiplication could be easily computed. We will discuss more of this in Section 3.6. This observation is further applied to derive a Secure Inverse of Matrix Sum Protocol using the formula $PP^{-1}(A+B)^{-1} = (A+B)^{-1}$. The overall process involves two SMM operations.

3 Embedding Caching in Data Mining Algorithms

In this section, we investigate how one can apply caching to improve the computational efficiency of PPDM algorithms. We analyze the efficiency improvement and memory required to cache data that are reused repeatedly.

3.1 Secure Multi-party Sum If the same dataset that is held by multiple parties are repeatedly used in a data mining algorithm, we can simply cache the results that were computed before. Although the result caching of the SMS protocol is rather straightforward, it improves the computation when there are new parties joining the system. The original set of parties may be conceptually treated as a *single party* and the SMS protocol can be applied in the usual manner to all parties, using the cached results of the *single party*.

Analysis The intermediate result caching of the SMS protocol reduces the amount of network messages and the computational overheads of each party. The memory used for caching is relatively small and thus negligible for each party involved. However, if only one or a few parties are in the system, it may not be very secure to use this caching technique. In this case, we may have to perform the SMS protocol anew.

3.2 Privacy-Preserving Cooperative Linear System of Equations Protocol With reference to the earlier description in Section 2.3, we want to compute the value of $P\,(M_1 + M_2)\,Q$, where M_1 (respectively M_2) is held privately by Party A (respectively B). Matrices P and Q are randomly generated matrices known to Party B only. By decomposing M_1 and M_2 into several sub-matrices by both parties, they exchange information securely using the 1-out-of-N Oblivious Transfer protocol [11]. The overall result follows.

As illustrated in Algorithm 1, the result caching of this protocol can be applied on matrix sequences

Algorithm 1 Private Evaluation of $M\prime = P\left(M_1 + M_2\right)Q$.

```
Input: Alice has a matrix $M_1$, and Bob has a matrix $M_2$ and two random matrices $P$ and $Q$.
Output: $M\prime = P\left(M_1 + M_2\right)Q$
1: Alice and Bob agree on two large numbers $p$ and $m$.
2: Alice generates $m$ random matrices $X_1, \ldots, X_m$, such that $M_1 = X_1 + \cdots + X_m$.
3: Bob generates $m$ random matrices $Y_1, \ldots, Y_m$, such that $M_2 = Y_1 + \cdots + Y_m$.
4: for each $j = 1, \ldots, m$, Alice and Bob conduct the following sub-steps: do
5:     Alice sends the following sequence to Bob: $(H_1, \ldots, H_p)$ where for a secret $1 \leqslant k \leqslant p, H_k = X_j$; the rest of the sequence are random matrices. $k$ is a secret random number known only by Alice, namely Bob does not know the position of $X_j$ in the whole sequence. Bob could cache the matrix sequence $(H_1, \ldots, H_p)$ for future convenience during this process.
6:     Bob computes $P\left(H_i + Y_j\right)Q + R_j$ for each $i = 1, \ldots, p$, where $R_j$ is a random matrix.
7:     Using the 1-out-of-$N$ Oblivious Transfer protocol, Alice gets back the result of $G_j = P\left(H_k + Y_j\right)Q + R_j = P\left(X_j + Y_j\right)Q + R_j$. Alice could cache the matrix $G_j$ for future convenience during this process.
8: end for
9: Bob sends $\Sigma_{j=1}^{m} R_j$ to Alice.
10: Alice computes $M\prime = \Sigma_{j=1}^{m}\left(P\left(X_j + Y_j\right)Q + R_j\right) - \Sigma_{j=1}^{m} R_j = P\left(M_1 + M_2\right)Q$.
```

H_js in Party B and G_js in Party A. In the case when the matrix input from B is modified from M_2 to $M_2^{'}$ in future, B can choose any ℓ ($\ell \in [1, m]$) out of m Y_js for suitable adjustment. The total number ℓ and the index of matrices can be chosen and selected arbitrarily, depending on the security and difficulty level for a party to speculate the information. In other words, the number of matrices that Party B choose to modify greatly affects the data privacy of its own data. The more matrices that Party B modifies, the lower is the possibility of Party A's ability to speculate the information. However, this result caching scheme is relatively secure for $\ell \geqslant 2$ in general.

For example, suppose B decides to modify the value of matrices Y_1 & Y_2 to $Y_1^{'}$ & $Y_2^{'}$ such that the equation $M_2^{'} = Y_1^{'} + Y_2^{'} + Y_3 + \cdots + Y_m$ is satisfied. Then B will only re-compute the sequence $G_1^{'}$ & $G_2^{'}$ based on the sequence H_1 & H_2 cached. After Party A receives the updated matrix sequences $G_1^{'}$ & $G_2^{'}$, it simply obtains the result back with a secret random number k and performs the rest of the computations with sequence $G_3, G_4, \ldots, G_m$ cached in the last iteration. Hence, before applying this caching approach, we would like to choose the party that has a higher probability of changing the matrix input, such as Party B above. In the case of two parties changing their inputs together, A could also modify its matrix sequence X_i in same manner as well.

Analysis The intermediate result caching on this algorithm greatly reduces the computational complexity in terms of the number of oblivious transfers as well as matrix multiplications. By caching the previous result, the total number of oblivious transfers is reduced from m to 2, following the above description. Also, the number of matrix multiplications carried out is $2m$ instead of m^2, a significant reduction in total computation time. This also reduces the overall network communication cost.

Memory usage for the caching scheme depends on the number of matrices and their dimensions. If we assume $m - \ell$ out of m matrices are maintained in subsequent iterations and every matrix dimension is $d \times d$, the minimum memory (in bytes) required for Alice is $(m - \ell) \times d^2 \times b$ bytes, where b denotes the number of bytes for storing a single element in a matrix. As Bob stores a series of matrix sequences, the overall memory required by him is relatively large compared to Alice; it requires $(m - \ell) \times d^2 \times b \times p$ bytes in total.

Data privacy is preserved during caching. Given that the original protocol is secure, Bob would not be able to derive any solvable linear system of equations by changing its matrix to probe Alice's private matrix. This is true for Alice as well. Both parties agree on the number of matrices in the sequence that they are going to modify. Theoretically speaking, more number of matrices modified leads to better security. In general, two matrices are more than sufficient to guarantee the privacy of both parties, as it can be proved that both parties are not able to establish any linear system of equations to derive the other party's private data. If this algorithm with caching is executed many times, two parties could come to a consensus to modify different portions of their matrix sequence based on the cached results in different iterations to avoid the derivation of any solvable linear system of systems by any party.

Depending on pre-protocol agreement, Party A may cache $\sum_{j=1}^{m} R_j$ if Party B maintains the same $\Sigma_{j=1}^{m} R_j$ but with modifications on corresponding items in the random number series R_j in each it-

Table 1: Data partitions among 3 parties

TID	A_1	A_2	B_1	C_1	*Class*
T1	1	0	1	0	1
T2	1	1	0	1	1
T3	0	1	0	1	1
T4	1	0	1	1	0
T5	1	1	1	1	0

eration. Even though the improvement is minor, it reduces network communication cost if the network data transfer overhead is relatively high.

3.3 Privacy-Preserving Association Rule Mining As described in Section 2.4, multiple parties performing the *Apriori* algorithm iteratively generate frequent itemsets using PPARM protocol. We observe that intermediate result caching can be applied throughout the entire algorithm.

Let us elaborate this process using the example in Table 1. Suppose after the first round of computations, attributes A_1 and A_2 are held by Party A, attribute B_1 is held by Party B, and attribute C_1 is held by Party C. In the next iteration, A and B compute the frequency counts of itemsets $\{A_1, B_1\}$ and $\{A_2, B_1\}$ collaboratively. Within each iteration, one party caches the encrypted sequence from the other party. In this case, B sends the encrypted sequence of B_1 to A, and Party A caches the sequence and use it throughout the following computations.

Consider the case when the frequency counts of all itemsets generated in the 2nd iteration are greater than the support threshold. The three parties will further proceed to compute the frequency count of $\{A_1, B_1, C_1\}$ and $\{A_2, B_1, C_1\}$ in the next iteration. Consider the computational process $\{A_1, B_1, C_1\}$. Party C requires the encrypted vector $\{A_2, B_1\}$, which was computed in the previous round. If Party A or B properly caches the sequence, the overall number of cryptography computations is greatly reduced.

Analysis The exact number of SSP operations trimmed by caching is dependent on the number of frequent itemsets involving a particular input vector generated by the *Apriori* algorithm. Let us assume in one iteration, the number of frequent itemsets involving an input data vector $\vec{V}$ is k. The length of $\vec{V}$ is ℓ. If proper caching is applied, the number of cryptography computations is reduced from $(k+1)\ell$ to $k + \ell$.

To apply this caching scheme, additional memory blocks for storing completely encrypted vectors from previous iterations are required. For example, suppose at iteration i there are c candidate frequent itemsets satisfying the support threshold. Then at iteration $i+1$, the overall caching memory required is $c \times \ell$ in the worst case, where ℓ denotes the size of each vector. In practice, many vectors overlapped one another, it is a lot less than $c \times \ell$. As can be observed, the memory required is sequentially reducing, as the number of frequent itemsets is decreasing. As a result, the system should reserve the necessary amount of memory for caching according to the number of single attributes that satisfy the support threshold during the first iteration, as the number of frequent itemsets is maximum at that time.

Security is not an issue in this algorithm as long as the number of tuples is larger than the dimension of dataset (which is true in real applications) as both parties are not able to build a linear system to derive the private data of other parties. This problem is associated with the original PPARM algorithm without caching. If the original PPARM protocol is proved to be secure with input datasets, applying the caching scheme is secure as we only cache intermediate results that are proven to be secure.

3.4 Privacy-Preserving Decision Tree Induction The part of PPID3 that requires SSP is the computation of the information gain (i.e., entropy) in order to determine the best splitting attribute. Intermediate result caching scheme can be applied throughout the entire algorithm in a similar manner as PPARM as the input data are all strictly binary sequences. In this section, we introduce three caching strategies that improves the efficiency of the algorithm. Referring to the data in Table 1, suppose the class attribute is owned by all parties and the algorithm proceeds to node $(B_1 = 0)$. Let n be the domain size of the class attribute and k be the number of values of the observed attributes (n and k are equal to 2 in this case). If we want to compute the information gain of attribute A_1, we evaluate:

$$\begin{aligned}&\mathit{Entropy}(D_{B_1=0}, A_1 = 0)\\ &= -\frac{P_{Class=0}}{|D_{A_1=0}|}\log\frac{P_{Class=0}}{|D_{A_1=0}|} - \frac{P_{Class=1}}{|D_{A_1=0}|}\log\frac{P_{Class=1}}{|D_{A_1=0}|}\end{aligned}$$

and

$$\begin{aligned}&\mathit{Entropy}(D_{B_1=0}, A_1 = 1)\\ &= -\frac{P_{Class=0}}{|D_{A_1=1}|}\log\frac{P_{Class=0}}{|D_{A_1=1}|} - \frac{P_{Class=1}}{|D_{A_1=1}|}\log\frac{P_{Class=1}}{|D_{A_1=1}|}\end{aligned}$$

where $P_{Class=0}$ (respectively $P_{Class=1}$) refers to the probability of $Class = 0$ (respectively $Class = 1$). All the computations of the above entropy computations are within the domain $B_1 = 0$. To compute the probability of every possible values of the class attribute involves a SSP computation. In total, for every observed attribute, $n \times k = 4$ SSP operations involving the same input vector B_1 are carried out. By caching vector B_1, we only need to send this vector once.

Party A has another attribute A_2 in addition to A_1. A 2nd caching strategy is applicable when the system computes the information gain of attribute A_2. As this computation is similar to the process described above, it also requires the vector on attribute B_1. So caching on vector B_1 is still applicable to other attributes for further reducing the cryptographic computational overheads.

A 3rd caching strategy is based on the iterative characteristic of the ID3 algorithm. It is known that the SSP protocol executed in each iteration always depends on its result from the previous iterations (except the first iteration). Continuing from the example above, assume that in the second iteration, we choose attribute A_2 as the splitting attribute and now we are at tree node $(B_1 = 0, A_2 = 0)$. The information gain at this point, as stated previously, is obtained by performing SSP between the B_1 vector owned by B and the A_2 vector owned by A. In this process, if we assume Party B initiates the SSP protocol, Party A retrieves a vector denoted as $\vec{V} = A_2 \times B_1$ in encrypted form.

The next step of the protocol is to compute the information gain on attribute C_1 of party C, which again requires a SSP computation involving all three parties. Now, the encrypted vector $\vec{V}$ from the previous iteration cached by Party A can be treated as the input to Party C directly. Eventually, after the protocol is completed, the scalar product of all three vectors $A_2 \cdot B_1 \cdot C_1$ can be computed easily.

Analysis The number of cryptographic computations reduced via caching depends on characteristics of the constructed decision tree. However, the improvement is predictably significant as almost all repeated cryptographic computations are eliminated. Similar to the caching scheme applied on PPARM algorithm, the security of the overall algorithm is guaranteed with this caching scheme.

The memory requirement for caching scheme on PPID3 would be relatively large compared to PPARM, as splitting attribute for every ancestor should be cached in order to implement the third caching strategy that is to efficiently compute the information gain for child and sibling node. Suppose d is the depth of the tree and n represents the number of data. If we denote k as the approximate size of an encrypted number (in byte). In total, we will need memory of $d \times n \times k$ bytes to implement the third caching strategy. In addition, for every iteration to efficiently compute information gain on every attribute, we also need $n \times k$ bytes memory for implementing the first and second caching strategy. However, this memory can be released when the party has not had any attribute to be considered in that iteration. Therefore, in the worst case scenario we will require additional $(d + 1) \times n \times k$ bytes to implement all caching strategies.

Algorithm 2 Privacy Preserving k-means Clustering Protocol.

Input: Database D consisting of n data objects arbitrarily partitioned by Alice and Bob.
Integer k denotes the number of cluster centers.
Output: Assignment of the cluster centers to input data objects.

```
1: Randomly select $k$ objects from $D$ as initial cluster centers $\mu\prime_1 \dots \mu\prime_k$.
2: Randomly generate all cluster center shares for Alice and Bob:
   Alice's share = $\left(\alpha_1^A \dots \alpha_k^A\right)$
   Bob's share = $\left(\alpha_1^B \dots \alpha_k^B\right)$
3: **repeat**
4:     $\left(\mu_1^A \dots \mu_k^A\right) = \left(\alpha_1^A \dots \alpha_k^A\right)$
       $\left(\mu_1^B \dots \mu_k^B\right) = \left(\alpha_1^B \dots \alpha_k^B\right)$
5:     **for** each $d_i$ in $D$ **do**
6:         Compute and assign $d_i$ to the closest cluster.
           **Alice (respectively Bob) could cache the vector sequence $(\mu_1^B \dots \mu_k^B)$ (respectively $(\mu_1^A \dots \mu_k^A)$) for convenience during this process.**
7:     **end for**
8:     Alice and Bob securely recompute random shares of the centers of the $k$ clusters as $\left(\alpha_1^A \dots \alpha_k^A\right)$ and $\left(\alpha_1^B \dots \alpha_k^B\right)$ respectively.
9: **until** $\left(\mu_1^A + \mu_1^B, \cdots, \mu_k^A + \mu_k^B\right)$ is close enough to $\left(\alpha_1^A + \alpha_1^B, \cdots, \alpha_k^A + \alpha_k^B\right)$
```

3.5 Privacy-Preserving k-means Clustering

As described in Section 2.6, intermediate result caching can be applied. Algorithm 2 illustrates the PPKC protocol incorporating intermediate result caching. The core portion of the PPKC algorithm is to compute the distance between a data point and every cluster center. This process involves SSP computations among different parties iteratively. Consider a data point d_i which is represented by ℓ distinct attributes $d_i = (x_{i,1}, x_{i,2}, \dots, x_{i,\ell})$. Without loss of generality, assume $x_{i,1}, \dots, x_{i,s}$ belong to Party A and the rest of the $\ell - s$ attributes belong to Party B. The equation used to compute the distance $\texttt{dist}\,(d_i, \mu_j)$ between the data input d_i and j-th cluster mean μ_j is as follows:

$$\begin{aligned}
&(\texttt{dist}\,(d_i, \mu_j))^2 \\
&= \left(x_{i,1} - \left(\mu_{j,1}^A + \mu_{j,1}^B\right)\right)^2 + \cdots \left(x_{i,\ell} - \left(\mu_{j,\ell}^A + \mu_{j,\ell}^B\right)\right)^2 \\
&= \sum_{m=1}^{\ell} x_{i,m}^2 + \sum_{m=1}^{\ell} \left(\mu_{j,m}^A\right)^2 + \sum_{m=1}^{\ell} \left(\mu_{j,m}^B\right)^2 \\
&- 2\sum_{m=1}^{\ell} \mu_{j,m}^A x_{i,m} - 2\sum_{m=1}^{l} x_{i,m}\mu_{j,m}^B \\
&+ 2\sum_{m=1}^{\ell} \mu_{j,m}^A \mu_{j,m}^B
\end{aligned}$$

as $\mu_{j,t} = \mu_{j,t}^A + \mu_{j,t}^B$, where $\mu_{j,t}^A$ (respectively $\mu_{j,t}^B$) represents the t-th item of A's (respectively B's) share of the j-th cluster center. The first three terms can be computed individually by Party A and B but the last three terms need a SSP protocol between the two parties. Within the same iteration, $\mu_{j,t}^A$ and $\mu_{j,t}^B$ are constant until all data inputs have been successfully examined and assigned. Therefore, they are amenable to caching. After they exchange the encrypted vectors of all cluster centers at the beginning of each iteration, the distance between any input data object and every cluster center is easily obtained. In other words, the cryptographic overheads of the last three items are reduced to two decryption operations as the result of $\sum_{m=1}^{\ell} \mu_{j,m}^A \mu_{j,m}^B$ can be reused within the same iteration for the same j value.

Caching efficiency involving cluster centers is further seen in the termination condition of the algorithm. At the end of every iteration, cluster centers are updated accordingly. The algorithm terminates when there are no substantial improvement in reducing the error in approximation, which is measured in terms of the Euclidean distance between the original and updated cluster centers as follows:

$$\begin{aligned}
&\left(\texttt{dist}(\mu_j^{old}, \mu_j^{new})\right)^2 \\
&= \left(\texttt{dist}(\mu_j^{A,old} + \mu_j^{B,old}, \mu_j^{A,new} + \mu_j^{B,old})\right)^2 \\
&= \sum_{m=1}^{\ell} \left(\mu_{j,m}^{A,new} + \mu_{j,m}^{B,new}\right)^2 + \sum_{m=1}^{\ell} \left(\mu_{j,m}^{A,old} + \mu_{j,m}^{B,old}\right)^2 \\
&- 2\sum_{m=1}^{\ell} \left(\mu_{j,m}^{A,old} + \mu_{j,m}^{A,old}\right)\left(\mu_{j,m}^{A,new} + \mu_{j,m}^{A,new}\right) \\
&= \sum_{m=1}^{\ell} \left(\mu_{j,m}^{A,new} + \mu_{j,m}^{B,new}\right)^2 + \sum_{m=1}^{\ell} \left(\mu_{j,m}^{A,old} + \mu_{j,m}^{B,old}\right)^2 \\
&- 2\sum_{m=1}^{\ell} \mu_{j,m}^{A,old}\mu_{j,m}^{A,new} - 2\sum_{m=1}^{\ell} \mu_{j,m}^{B,old}\mu_{j,m}^{B,new} \\
&- 2\sum_{m=1}^{\ell} \mu_{j,m}^{A,old}\mu_{j,m}^{B,new} - 2\sum_{m=1}^{\ell} \mu_{j,m}^{B,old}\mu_{j,m}^{A,new}
\end{aligned}$$

and the notation *old* (respectively *new*) simply stands for the original (respectively updated) cluster center. When result caching is applied, we observe that only the first item requires a SSP while all others items require a decryption instead. As a result, the total number of cryptographic computation is reduced significantly.

Analysis As stated previously, caching is applicable to the clustering centers. Let us assume the input dataset to the PPKC system contains n input tuples distributed over ℓ number of attributes. At every iteration, before we assign each data input to the nearest cluster center, the distances between any two of them are computed. Thus, there are $3 \times n \times k$ SSP operations in total. To test for convergence, the distance (also referred as error rate) between the old and new cluster centers are computed. This requires 4 SSP computations. In total, there are $4 \times k$ SSP operations. Based on these observations, the total number of encryptions (S_{enc}) are:

$$S_{enc} = t \times (3 \times n \times k + 4 \times k) \times \ell$$

where k represents the total number of clusters and t is the number of iterations required before it reaches convergency.

As elaborated above, both parties are required to cache the encrypted vectors of all other party's share of cluster centers. This means that both parties need to encrypt all their cluster centers of length equals to $k \times \ell$ concurrently. Furthermore, after both parties obtain the new cluster center, they need to compute the distance between the previous cluster center and the new cluster center. In this computation, Bob needs to cache the encrypted value of the partial new cluster center held by Alice and so does Alice. So, in the first iteration, we need $2 \times k \times \ell$ encryptions. However, this new cluster center will be used again for the next iteration if the iteration is not the last to compute the distance between all cluster centers and data tuples. This is why for the following iterations, we need only $1 \times k \times \ell$ encryptions; thus, yielding a total of $(t+1) \times k \times \ell$ encryptions. After intermediate result caching scheme is applied, the number of encryption computations is essentially reduced to

$$S_{enc} = (t + 1) \times k \times \ell$$

. The number of decryptions is ignored as it remains constant regardless of whether result caching scheme is applied.

The extra memory required for caching is $k \times \ell$ vectors at both sites, as both parties use it hold the share held by the other party for every cluster center. If one encrypted number requires b bytes, both parties reserve $k \times \ell \times b$ bytes in advance to perform caching. There is no data privacy violation when caching is used as the private data will always be kept by the corresponding party. The information they exchanged is simply the shares of every cluster center, which is considered relatively less sensitive data.

3.6 Secure Matrix Multiplication & Inverse of Matrix Sum Computation To show the generalizable feature of the proposed technique, we include the efficient secure matrix multiplication protocol here as well [7]. And we also explore the secure inverse of matrix sum computation using embedded caching technique here.

Let us denote $A_1, A_2, \ldots, A_m$ as m row vectors of A and $B_1, B_2, \ldots, B_n$ as n column vectors of B. Recall the definition of matrix multiplication:

$$M = AB = \begin{bmatrix} A_1 \cdot B_1 & A_1 \cdot B_2 & \cdots & A_1 \cdot B_n \\ A_2 \cdot B_1 & A_2 \cdot B_2 & \cdots & A_2 \cdot B_n \\ \vdots & \vdots & \ddots & \vdots \\ A_m \cdot B_1 & A_m \cdot B_2 & \cdots & A_m \cdot B_n \end{bmatrix}$$

Essentially, every element in M is a scalar product of two vectors A_i and B_j ($i \in [1, m], j \in [1, n]$), which can be computed using any SSP protocol introduced in Section 2.1. To compute every element of every row i (respectively column j), Party B (respectively A) caches the encrypted vector of A_i (respectively B_j) sent by Party A (respectively B). The way we compute the matrix M row-wise or column-wise depends on the matrix dimension. In order to increase the efficiency of result caching, we always choose the loop with less iteration as the outer loop.

Efficient secure matrix multiplication is used to obtain the inverse of matrix sum as discussed in Algorithm3. In the secure inverse of matrix sum, Alice and Bob each hold private $d \times d$ matrix A and B respectively. They want to compute the inverse of $A + B$.

In Steps 2 and 5 of this protocol, the multiplication between two matrices is performed. We also notice that they are over the same matrix P. Thus, intermediate result caching is applied to P. However, take note that in these two matrix multiplications, P is involved in different ways. It is first passed column-wise and during the second multiplication, every row of P is exchanged among parties. Clearly, in Step 2, if we let Party B initiate the SSP mentioned in Section 2.1, B will send the entire matrix P in encrypted form to A before executing this protocol.

Analysis For matrix multiplication, there are n^2 encryptions and n decryptions in the computation process in total at the cryptographic level. This is more effective than the one without caching (n^3 encryptions and n decryptions).

Algorithm 3 Secure Inverse of Matrix Sum Protocol.

```
Input: Private $n \times n$ matrix $A$ and $B$ holding by Party A and B respectively.
Output: Private $n \times n$ matrix $M_a$ and $M_b$ holding by Party A and B respectively where $M_a + M_b = (A + B)^{-1}$.
1: B randomly generates a non-singular $n \times n$ matrix $P$.
2: A and B jointly perform secure matrix multiplication to compute $AP$, at end of which, A and B would hold value $S_a$ and $S_b$ correspondingly such that $S_a + S_b = AP$. **Party A could cache the matrix $P$ for future references during this process.**
3: B send $S_b + BP$ to A.
4: A computes $S_a + S_b + BP$ which equals to $(A + B)P$, and then compute inverse $P^{-1}(A + B)^{-1}$.
5: After B and A jointly perform secure matrix multiplication on $P$ and $P^{-1}(A + B)^{-1}$, they would hold value $M_b$ and $M_a$ where $M_a + M_b = PP^{-1}(A + B)^{-1} = (A + B)^{-1}$.
```

The memory used for this caching scheme would be relative small, simply the size of a $n \times n$ matrix. If we denote k as the memory (in byte) for holding one encrypted number, the total memory required is $n \times n \times k$ bytes.

Data privacy during matrix multiplication is preserved provided the SSP is secure, as in the process, Party A only sees an encrypted $n \times n$ matrix from B if we assume Party B initiates the protocol. For the inverse of matrix sum computation, as it is built upon several secure building blocks, we can easily prove that there is no data leakage and privacy disclose to other parties.

4 Experimental Evaluations

In the following sections, we present experiment results of incorporating caching. The system is implemented in Java using Netbeans 6.1 and the platform is a Windows XP system running on an Intel Pentium 4 3.40 GHz processor with 1 GB of main memory. The sample dataset is the Nurse dataset [1] from the UCI depository. This dataset has 9 attributes and 12,960 tuples. We perform some preprocessing on the data, such as converting it to binary data. The communication network is simulated by passing data in a self loop. We compared the total number of cryptographic operations required for PPDM algorithms with and without incorporating caching.

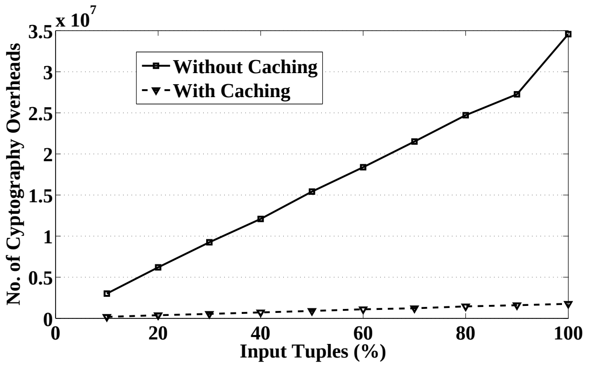

Figure 1: Cryptographic computations for different number of tuples in association rule mining.

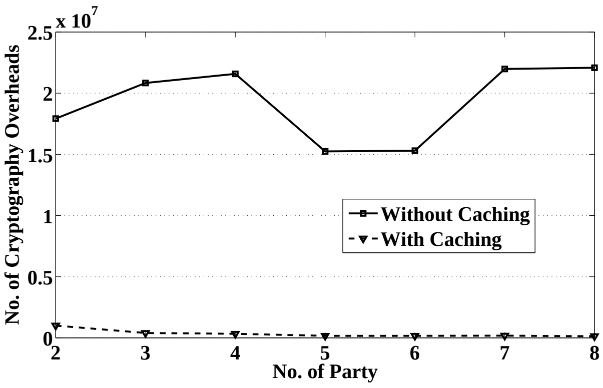

Figure 2: Cryptographic computation for different numbers of party in association rule mining.

4.1 Privacy-Preserving Association Rule Mining Figure 1 illustrates the effects of caching efficiency versus the number of tuples. The system first extracts some portions of the entire dataset. Then the algorithm is run several times with different number of input tuples. All of the tuples are selected randomly according to uniform distribution to avoid using clustered data for each experiment. As expected, when the number of data tuples increases, we have more itemsets that satisfy the support threshold. Thus, more SSP and cryptographic computations are involved. With caching, similar behavior appears, but at a much slower rate. Hence, the overall performance of the PPDM system greatly improved, especially when the dataset is very large, as intermediate result caching provides a more effective way to handle this situation.

Figure 2 elaborates the relationship between the number of parties in the PPDM versus the amount of cryptographic computations. We splitted all attributes in the Nurse dataset randomly among different parties. The number of SSP operations does not fully depend on the number of parties, but more so on the frequent itemsets generated at every iteration. If the itemsets generated are fully owned by the same party, we will not need any SSP computation. As a result, the total number of cryptographic computations is affected by the partition scheme that parties choose over the entire dataset. The behavior of the graph does not show much pattern. Instead, it fluctuates within a certain range.

The SSP protocol (Section 2.1) that we use for this experiment does not require any additional encryptions when the number of parties increases. As shown, for the two-party setting, the SSP protocol needs $n+1$ encryptions where n is the vector length. In our case, vector length is the number of tuples involved in the experiment. In the three-party scenario, the protocol needs only $n+3$ encryptions in total. Additional encryptions do not affect the overall performance if n is substantially large.

Still, when caching is applied, we observe major improvement in the overall processing speed of the PPDM system. The total number of cryptographic computations after we apply caching is about $\frac{1}{10}$ of the original protocol.

In Figure 3, we address the impact to caching efficiency under different support thresholds. In this experiment, we choose different threshold percentages in the range between 1% and 25%. The support threshold is defined as the number of total tuples involved in the experiment times the percentage. We randomly choose 6,473 tuples from the Nurse dataset as the input data. In general, there is a negative correlation between the support threshold value and total number of cryptographic computations. A higher support threshold essentially reduces the number of frequent itemsets. Hence, it reduces the number of iterations and SSP computations required.

Meanwhile, as we can observe from the figure, the intermediate result caching scheme indeed greatly improves system efficiency. This is especially true when the support threshold is relatively low, while a comparably larger amount of SSP computations is required.

4.2 Privacy-Preserving Decision Tree In Figure 4, we illustrate the relationship between the overall number of cryptographic computations and the number of input tuples. Like experiment on PPARM, the system feeds the program with only a portion of the total Nurse dataset each time; thus, simulating different number of input tuples to the system. As observed, an increase in the total number of input tuples eventually leads to an increase in the complexity of constructing a decision tree classifier. However, as we can see from the diagram, caching shows better

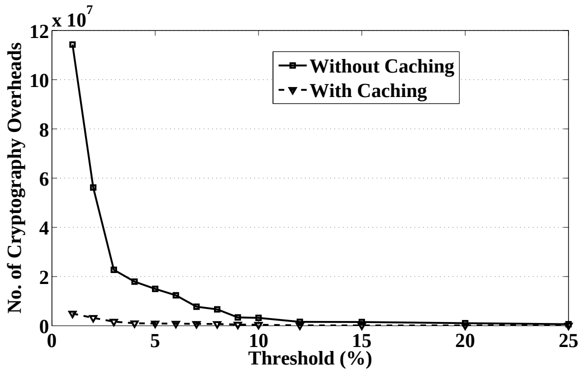

Figure 3: Cryptographic computations under different support thresholds for association rule mining.

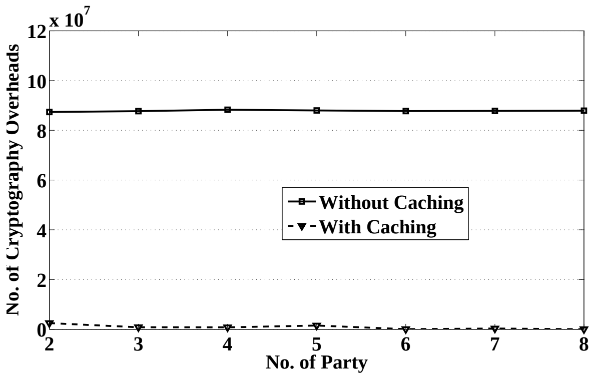

Figure 5: Cryptographic computations for different number of parties in decision tree induction.

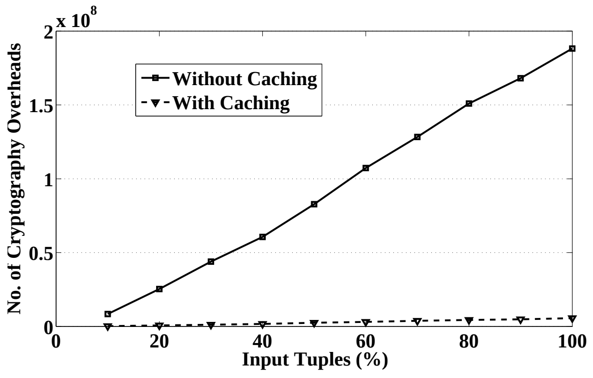

Figure 4: Cryptographic computations for different number of tuples in decision tree induction.

performance compared to the original protocol without caching irrespective of the number of tuples to the system. Since intermediate result caching mainly focuses on the reduction of the number of cryptographic computations, the more SSP operations are involved, the better is the caching scheme performance. In fact, experimental results indicate that caching technique works a lot better when the input dataset is large.

PPID3 algorithm is able to handle multi-party situations as well. The total number of parties involved in building a decision tree classifier is taken into account in Figure 5. In this figure, even with different number of parties, the outputs are quite stable with only minor fluctuations in both curves representing the cases with and without caching respectively. A comparison of the two results shows that the overall system performance is improved tremendously with intermediate result caching.

4.3 Privacy-Preserving k-Means Clustering in Arbitrarily Partitioned Data Unlike PPARM or PPID3 where classification execute on binary data inputs, PPKC performs on any real-valued data. We use the Nurse dataset as input. The total number of cryptographic computations can be calculated explicitly. Hence, the efficiency of intermediate result caching is easy to observe.

Experimental results on the PPKC algorithm provides supporting evidence on previous computations. In Figure 6, we illustrate the impact of the number of cryptographic computations under different number of input tuples. Figure 7 shows the relationship between the number of clusters and the total number of cryptographic computation required. The diagram in Figure 8 shows the comparison between different number of cryptographic computations according to the different number of iterations in both intermediate result caching scheme applied and original algorithm. All three graphs indicate the expected linear outputs irrespective of the underlying input parameter that fit the previous computation formulas. After result caching scheme is applied, the number of decryptions is the same as the original algorithm. However, the total number of encryptions is reduced significantly. As a result, the larger the input dataset is, the more efficient and effective is the intermediate result caching scheme. In practical applications, if we have an extremely large dataset, the caching technique greatly improves the overall processing speed of the system.

4.4 Secure Matrix Multiplication We implemented the SMM protocol to illustrate caching efficiency in C# language under Microsoft Visual Studio 2005 environment. The experiments are tested on Windows XP operating system with Intel Pentium 4 3.40GHz and 3GB memory. The input matrices to the system are of $m \times N$ and $N \times n$ dimension correspondingly. The values are generated randomly and distributed uniformly in a mathematical field $\mathbb{F}$.

Figure 9 illustrates the effect of total processing

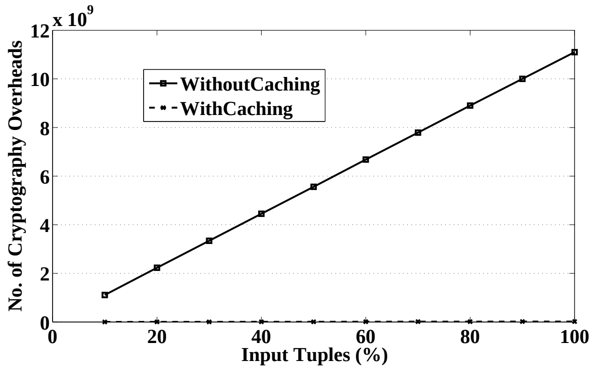

Figure 6: Cryptographic computations for different number of tuples in k-means clustering.

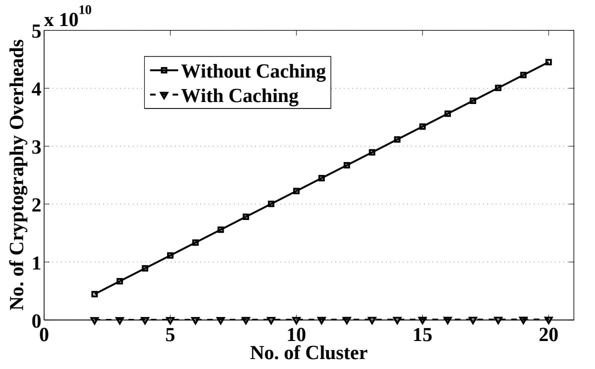

Figure 7: Cryptographic computations for different number of clusters for k-means clustering.

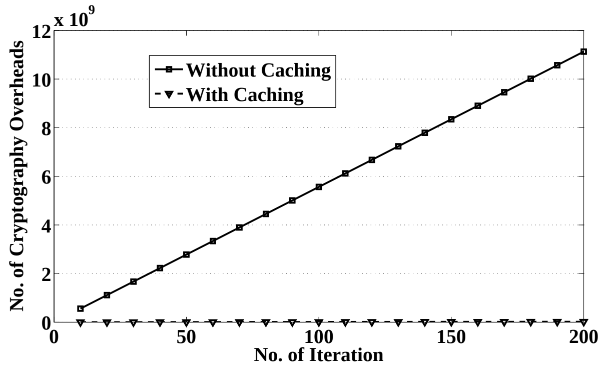

Figure 8: Cryptographic computations under different number of iterations for k-means clustering.

time with respect to different dimensions of input matrices. With result caching, the overall system performance is improved greatly. This is especially true when input matrices are large.

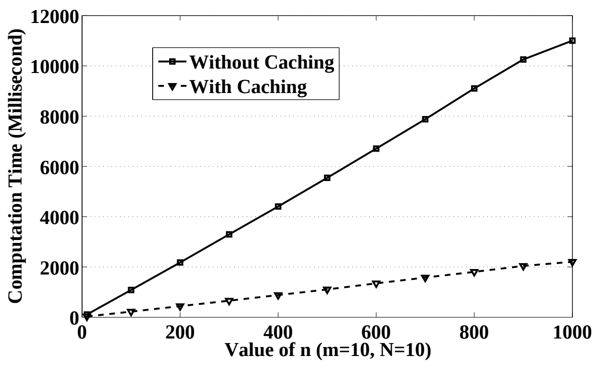

Figure 9: Processing time for different input matrix dimension n in secure matrix multiplication.

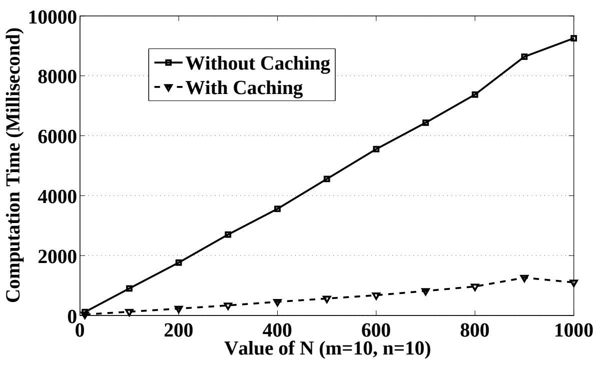

Figure 10: Processing time for different input matrix dimension N in secure matrix multiplication.

5 Conclusions

Secure computation is the most critical part of PPDM algorithms. Although it preserves the privacy of each party's data, it also incurs the majority of computational cost of algorithm execution. In this paper, we addressed the inefficiency issues in existing PPDM algorithm such as PPARM, PPID3, and PPKC and proposed an approach to resolve inefficiency using result caching.

We have implemented and experimentally evaluated our approach in PPDM algorithms and compared its performance with PPDM algorithms that do not incorporate caching. All of our experimental results show very high degree of reduction in number of cryptographic computations incurred.

References

[1] UCI machine learning repository: Nursery data set http://archive.ics.uci.edu/ml/datasets/nursery.

[2] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proceedings of the ACM international Conference on Management of Data*, pages 439–450, Dallas, Texas, United States, 2000.

[3] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, and M. Y. Zhu. Tools for privacy preserving distributed data mining. *In SIGKDD Explorations*, 4(2):28–34, December 2002.

[4] W. Du and M. J. Atallah. Privacy-preserving cooperative scientific computations. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, page 273, Washington, DC, USA, 2001. IEEE Computer Society.

[5] B. Goethals, S. Laur, H. Lipmaa, and T. Mielikainen. On private scalar product computation for privacy-preserving data mining. In *Proceedings of the 7th Annual International Conference in Information Security and Cryptology*, pages 104–120, Seoul, Korea, December 2–3 2004.

[6] S. Han and W. K. Ng. Privacy-preserving linear fisher discriminant analysis. In *Proceedings of the 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Osaka, Japan, May 2008.

[7] S. Han, W. K. Ng, and P. Yu. Privacy-preserving singular value decomposition. In *Proceedings of the 25th IEEE International Conference on Data Engineering*, Shanghai, China, April 2009.

[8] G. Jagannathan and R. N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *Proceedings of the 8th ACM International Conference on Knowledge Discovery in Data Mining*, pages 593–599, Chicago, Illinois, USA, 2005.

[9] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Advances in Cryptology*, volume 1880 of *Lecture Notes in Computer Science*, pages 36–53. Springer-Verlag, 2000.

[10] K. Liu and R. Wolff. International workshop on practical privacy-preserving data mining held in conjunction with siam sdm'08, 2008.

[11] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 245–254, Atlanta, Georgia, United States, 1999.

[12] H. Subramaniam, R. Wright, and Z. Yang. Experimental analysis of a privacy-preserving scalar product protocol. In *Proceedings of the Workshop on Secure Data Management (held in conjunction with VLDB'04)*, 2004.

[13] J. Vaidya and C. Clifton. Privacy preserving association rule mining in vertically partitioned data. In *Proceedings of the 8th ACM International Conference on Knowledge Discovery and Data Mining*, pages 639–644, Edmonton, Alberta, Canada, July 23-26 2002.

[14] J. Vaidya and C. Clifton. Privacy-preserving decision trees over vertically partitioned data. In *Proceedings of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Storrs, Connecticut, 2005. Springer.

[15] A. C. Yao. How to generate and exchange secrets. In *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.