Secure Frequent Pattern Mining by Fully Homomorphic Encryption with Ciphertext Packing

Hiroki Imabayashi, Yu Ishimaki, Akira Umayabara,
Hiroki Sato, and Hayato Yamana

Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan
{imabayashi, yuishi, uma, hsato, yamana}@yama.info.waseda.ac.jp

Abstract. We propose an efficient and secure frequent pattern mining protocol with fully homomorphic encryption (FHE). Nowadays, secure outsourcing of mining tasks to the cloud with FHE is gaining attentions. However, FHE execution leads to significant time and space complexities. P3CC, the first proposed secure protocol with FHE for frequent pattern mining, has these particular problems. It generates ciphertexts for each component in item-transaction data matrix, and executes numerous operations over the encrypted components. To address this issue, we propose efficient frequent pattern mining with ciphertext packing. By adopting the packing method, our scheme will require fewer ciphertexts and associated operations than P3CC, thus reducing both encryption and calculation times. We have also optimized its implementation by reusing previously produced results so as not to repeat calculations. Our experimental evaluation shows that the proposed scheme runs 430 times faster than P3CC, and uses 94.7% less memory with 10,000 transactions data.

Keywords: Ciphertext packing · Fully homomorphic encryption · Frequent pattern mining · Privacy preservation · Cloud computing

1 Introduction

In the present era of big data, demand is increasing for outsourcing both data storage and calculations to the cloud. Although such outsourcing is convenient for users, there are security and privacy issues. Private information could be obtained maliciously by data "snooping" or covert monitoring. Thus, secure and privacy-preserving outsourced calculation has become indispensable, regardless of whether or not users trust the cloud.

In this paper, we focus on privacy-preserving data mining for outsourced calculations [7]. Previous research on such data mining is classified into three approaches: i) protecting input privacy, ii) protecting output privacy, and iii) cryptosystems. Protecting input privacy preserves the input data on the user side by abstraction, noise-addition or randomization [6, 16, 19, 21], while protecting output privacy preserves mining results on the cloud side by either noise-addition

or perturbation [3, 4]. The third approach, i.e., cryptosystems, preserves privacy by executing mining algorithms on encrypted data [11, 12]. Each approach has its advantages and disadvantages. While the computational costs of the input and output privacy approaches are smaller than those of cryptosystems, full input and output privacy cannot be guaranteed. Furthermore, mining result may become ambiguous when input or output privacy is used. Finally, cryptosystems require excessive computational time, albeit that they assure both secure computation and mining accuracy. Based on the above considerations, we chose to adopt a cryptosystem to assure the full privacy.

A fully homomorphic cryptosystem is one that handles unlimited numbers of multiplications and additions of ciphertexts. Gentry [8] proposed Fully Homomorphic Encryption (FHE), which has been widely adopted by many data-mining researches for statistical calculations [15], machine learning algorithms [9, 13], and the frequent pattern mining [11, 14].

However, these FHE applications suffer from the limitations of the computational resources such as memory and storage. They also take excessively long to execute because of large size ciphertexts and the significant number of associated operations. The data mining process itself also involves a high computational cost when handling large data sets. As a secure mining protocol, Liu et al. [14] proposed the P3CC frequent pattern mining scheme over FHE, which was implemented over DGHV integer-based FHE by van Dijk et al. [20]. It encrypts plaintexts component-wise in the item-transaction matrix data, and then applies addition or multiplication operation to each ciphertext individually. Therefore, the total number of ciphertexts increases linearly with the matrix size, which results in the excessive memory/storage usage, communication costs, and the operational costs for encrypted data.

To solve the above problems, it is essential to execute mining tasks with both a reduced size of ciphertext and fewer encrypted-data operations. In order to realize this, we adopt the polynomial Chinese Reminder Theorem (CRT) packing method proposed by Smart and Vercauteren [17, 18] with the Ring Learning With Errors (RLWE)-based BGV scheme [5]. With the packing method, we are able to pack multiple plaintexts into a ciphertext, followed by a parallelization of its element-wise vector multiplication. In comparison with P3CC, this results in smaller ciphertexts overall and fewer operations.

Our contribution is threefold. i) To the best of our knowledge, this is the first implementation of the frequent pattern mining constructed with the FHE packing method. ii) Our algorithm is optimized to pack the components column-wise in the item-transaction matrix data to reduce the number of ciphertexts and associated operations. Here, we define N as the number of transactions (i.e., the number of rows in the item-transaction matrix), and ℓ as the slot size (i.e., a ciphertext packs ℓ components of an item-column). In our algorithm, the number of ciphertexts required to pack all the components of an item column decreases from N to $\lceil N/\ell \rceil$, and hence the number of operations over all ciphertexts also decreases from N to $\lceil N/\ell \rceil$. iii) Both parallelization and caching technique are adopted to speed up the execution: parallelization for file reading/writing, en-

cryptions and calculations for each support value for encrypted data, and caching the previous results of the element-wise vector multiplication. FHE requires numerous multiplications among ciphertexts during such a multiplicative process, thus caching technique will work effectively.

The rest of this paper is organized as follows. We review related work in Section 2, and then introduce the background to the RLWE-based FHE scheme and the P3CC protocol in Section 3. We propose our scheme for efficient frequent pattern mining in Section 4, followed by its experimental evaluation in Section 5. Lastly, we conclude this paper in Section 6.

2 Related Work

In this section, we discuss related research on data mining with cryptosystems. We then describe P3CC, which is the work most related to ours on frequent pattern mining.

Works on data mining with cryptosystems [11, 12] is classified into two categories: multi-party computation (MPC), and homomorphic encryption (HE). In MPC, Kapoor et al. [12] proposed an algorithm for pattern mining that targets distributed database while preserving privacy by MPC. In HE, Mohammed et al. [11] proposed a secure comparison technique with FHE in the case of two-party association rule mining. They then showed that MPC is not suitable for association rule mining due to its storage, communication, and computational limitations.

The Privacy Preserving Protocol for Counting Candidates (P3CC) by Liu et al. [14] is the work that is most related to ours. Liu et al. employed an integer-based FHE [20] by van Dijk et al. It uses component-wise encryption on all the individual binary-represented components in the item-transaction matrix data. Liu et al. proposed α-pattern uncertainty for security in frequent pattern mining. This method maps items to meaningless symbols, and then adds dummy itemsets to prevent identification. The limitations of P3CC are its time complexity and the availability of computational resources, i.e., memory and storage. In addition, P3CC time complexity depends linearly on the number of transactions because of its component-wise encryption scheme. As for the execution time of P3CC, it takes from 1,000 to 10,000 seconds even with 5,000 transactions, with the minimum support ranging from 10% to 60% [14].

3 Preliminaries

In this section, we explain four algorithms that are used in latter sections: i) the Apriori algorithm, which is one of the best-known frequent pattern mining algorithms, ii) the P3CC algorithm, iii) polynomial CRT packing, and iv) the TotalSum algorithm for summing up all the elements of CRT-represented ciphertexts.

3.1 Apriori Algorithm

Agrawal and Srikant [2] proposed the now well-known algorithm called Apriori for mining frequent patterns. The transaction database, which consists of a set of items such as in Fig. 1 (a), can be mapped to the bit-represented item-transaction matrix shown in Fig. 1 (b). We show both the formal model of frequent patterns (Definition 1) [1] and the Apriori procedure (Algorithm 1) [14].

Trans. ID	Item Set
T1	{I1, I3, I4}
T2	{I3, I5, I6}
T3	{I2, I4, I5, I6}
T4	{I1, I2, I5}
T5	{I3, I6}
T6	{I1, I4, I6}

(a) original database

Items / Trans	I1	I2	I3	I4	I5	I6
T1	1	0	1	1	0	0
T2	0	0	1	0	1	1
T3	0	1	0	1	1	1
T4	1	1	0	0	1	0
T5	0	0	1	0	0	1
T6	1	0	0	1	0	1

(b) item-transaction matrix database

Fig. 1: Item-transaction database

Definition 1. ***(Frequent Patterns)*** *Let $I = \{i_1, i_2, ..., i_m\}$ be a set of m non-identical items, and let T be a set of transactions. Each transaction $t \in T$ has a set of items from I, i.e., t is a subset of I, with $t[k] = 1$ if t contains i_k, and $t[k] = 0$ otherwise. Let a pattern p be a subset of I. We say that a transaction t satisfies p, if and only if $t[k] = 1$ for all items i_k in p. The support of p is equal to the number of transactions in T that satisfy p. We say that a pattern is frequent if and only if its support is equal to or greater than a given minimum threshold called minSup.*

Firstly, the Apriori algorithm sets the frequent patterns of unit length, as L_1 by counting each item's support (lines 1-4). It then obtains all the frequent patterns in the iteration (lines 6-13).

Secondly, Apriori generates the length-2 candidate itemset C_2 from the frequent itemset L_1 (line 7), e.g., it generates $C_2 = \{\{a, b\}, \{b, c\}, \{a, c\}\}$ from $L_1 = \{\{a\}, \{b\}, \{c\}\}$. Thirdly, Apriori counts each length-2 pattern's support (lines 8-10) to obtain a new frequent pattern itemset L_2 by comparing with minSup (line 11), e.g., it obtains $L_2 = \{\{a, b\}, \{b, c\}\}$ if only the support of items $\{a, c\}$ is lower than minSup. Lastly, Apriori joins L_2 to the set A (line 12), followed by the execution of lines 6-13 repeatedly until no more candidates are generated.

Function *countSupport* calculates each support of candidate $c \in C_{i+1}$ by executing the element-wise AND operations over the item columns of c, followed by summing up all bits. For example, suppose we count the support of $c = \{\mathbf{a}, \mathbf{b}\}$, where $\mathbf{a} = (1, 0, 1, 1, 0)^T$ and $\mathbf{b} = (1, 1, 0, 1, 1)^T$ are vectors, each of

whose elements represents whether the i^{th} transaction has **a** or **b**. We first generate $(1, 0, 0, 1, 0)$ by executing element-wise AND operations, and then obtain a support of 2 by counting all unitary bits.

Algorithm 1 Apriori(*I, TDB, minSup*) [14]

Input: Itemset I; Transaction database TDB; Minimum threshold $minSup$;
Output: Frequent pattern itemset A;

1: **for each** *candidate* $c \in I$ **do**
2: $c.support :=$ countSupport(c, TDB);
3: **end for**
4: $L_1 := \{c \in I \mid c.support \geq minSup\}$;
5: $A \leftarrow L_1$
6: **for** $(i = 1; \|L_i\| > 1; i \leftarrow i + 1)$ **do**
7: $C_{i+1} :=$ generateCandidatePatterns(L_i);
8: **for each** *candidate* $c \in C_{i+1}$ **do**
9: $c.support :=$ countSupport(c, TDB);
10: **end for**
11: $L_{i+1} := \{c \in C_{i+1} \mid c.support \geq minSup\}$;
12: $A \leftarrow L_{i+1}$;
13: **end for**
14: **return** A;

3.2 P3CC and α-Pattern Uncertainty

P3CC as proposed by Liu et al. [14] adopts the "α-pattern uncertainty" algorithm, which decreases the probability of information leakage to attackers during P3CC server-client communication. Since FHE does not support comparison over encrypted data, P3CC has to return intermediate results of frequent pattern mining to the client. This is to both decrypt and compare them when numeric comparisons are required between each itemset's support and *minSup*.

Along with Algorithm 1, P3CC works as follows. i) As a preparation step, the client generates both public and secret keys to encrypt the database. Then, the client sends both the public key and the encrypted database to the server. ii) The server calculates each item's support over the encrypted data (lines 1-3) followed by sending the encrypted results back to the client. Then, the client obtains frequent items by comparing with *minSup* after decrypting the results (line 4). iii) The client generates a new candidate itemset, and then sends it to the server (line 7). iv) The sever calculates each pattern's support (lines 8-10) over the encrypted data, and then sends the encrypted results back to the client. v) The client decrypts the results to obtain the counted supports, and then compares each itemset's support with *minSup* over plaintexts (line 11). For each length-$(i + 1)$ itemset, iterate processes iii), iv), and v).

During the multiple occurrences of server-client communication described above, there exists a security issue whereby the server can infer the important itemsets by snooping on the candidate patterns obtained from the client. To prevent this, α-pattern uncertainty limits the server's certainty about frequent patterns. In other words, α-pattern uncertainty lowers the probability of an attacker inferring frequent patterns by employing dummy patterns. In this paper, we assume the α-pattern uncertainty achieves a *Semi-honest* model, where the server tries to distinguish true patterns from dummy patterns while following the protocol. The α-pattern uncertainty ensures that the server cannot infer true patterns more than the probability α. We do not discuss the security analysis in detail since it is not our objective. See the work by Liu et al. for further details [14].

3.3 Polynomial CRT Packing over FHE

Smart and Vercauteren [17, 18] proposed the CRT packing method over FHE. This allows multiple plaintexts to be packed into one ciphertext, which results in fewer ciphertexts. The following two steps generate a ciphertext in the polynomial CRT representation: i) multiple plaintexts are encoded into a single polynomial, i.e., CRT packing, and ii) encrypting the polynomial generates a ciphertext.

A CRT-represented ciphertext generated from ℓ plaintexts can be considered as a vector consisting of ℓ slots, each of which contains one plaintext. Multiplication over the CRT-represented ciphertexts is performed slot-by-slot in parallel, i.e., element-wise vector multiplication. See the work by Smart and Vercauteren [17, 18] for the mathematical construction of polynomial CRT packing with FHE.

3.4 Total Summation over CRT-represented Ciphertext

With the polynomial CRT packing method [17, 18], the FHE scheme needs to handle a ciphertext encrypted from multiple plaintexts, i.e., CRT-represented ciphertext. The following TotalSums proposed by Halevi and Shoup [10] is used for summing up all slots of a CRT-represented ciphertext. It takes a ciphertext encrypted from $(v_1, v_2, \ldots, v_n)$ as its input, and outputs a ciphertext that encrypts $(u, u, \ldots, u)$, where $u = \Sigma_{k=1}^{n} v_k$. The procedure is shown in Algorithm 2. See the work by Halevi and Shoup for detailed explanation of the algorithm and the implementation [10].

Algorithm 2 TotalSums($\boldsymbol{v}$) [10]

Input: Encrypted array $\boldsymbol{v}$;
Output: Encrypted array $\boldsymbol{u}$;

1: $\boldsymbol{u} := \boldsymbol{v}$, $e := 1$, $n := \|\boldsymbol{v}\|$;
2: $k :=$ numBits(n); # number of bits in n, e.g., numBits(5)=3
3: **for** ($j := k-2; j \geq 0; j \leftarrow j-1$) **do**
4: $\quad \boldsymbol{u} \leftarrow \boldsymbol{u} + (\boldsymbol{u} \ggg e)$; # $\ggg$: rotate operation
5: $\quad e \leftarrow 2 \cdot e$;
6: $\quad b :=$ bit$_j(n)$; # j-th bit of n, with bit 0 of LSB
7: $\quad$ **if** ($b = 1$) **then**
8: $\quad\quad \boldsymbol{u} \leftarrow \boldsymbol{v} + (\boldsymbol{u} \ggg e)$;
9: $\quad\quad e \leftarrow e + 1$;
10: $\quad$ **end if**
11: **end for**
12: **return** $\boldsymbol{u}$

4 Efficient Frequent Pattern Mining Algorithm over FHE

In this section, we propose an efficient frequent pattern mining algorithm that uses Apriori over FHE. It has minimal time and space complexities, and uses the polynomial CRT packing method and our caching technique. To begin with, we prepare an item-transaction binary-represented matrix data as shown in Fig. 2. Each column and row contains transactions and items, respectively. We use N_{trans} as the number of transactions, N_{item} as the number of items, and ℓ as the slot size of a ciphertext.

Since P3CC [14] encrypts plaintexts individually for each components in the item-transaction matrix data (as circled with heavy lines in Fig. 2a), it uses significant storage space and accrues excessive operational costs for encrypted data. This is because component-wise encryption increases the total size of ciphertexts linearly with the matrix size. In particular, P3CC generates $N_{item} \times N_{trans}$ ciphertexts in total, which requires $\Sigma_{i=1}^{m} p_i N_{trans}^{i-1}$ times multiplications to count the supports of all patterns, where p_i is the number of length-i candidate itemsets, N_{trans}^{i-1} is the $(i-1)^{th}$ power of N_{trans}, and m is the maximum length of the candidate itemset.

To reduce the time and space complexities, it is essential to execute mining tasks with smaller ciphertexts and fewer associated operations. In order to achieve this, we tune Apriori over FHE in two ways. Firstly, we adopt polynomial CRT packing [17, 18], which not only reduces the total ciphertext size, but also enables element-wise vector multiplication over ciphertexts in parallel, i.e., batching. Secondly, the execution of our scheme is accelerated by caching the previous results of the element-wise vector multiplication.

4.1 Polynomial CRT Packing and Batching

To reduce the time and space complexities of Apriori algorithm over FHE, polynomial CRT packing is adopted. We first port the framework of FHE from the P3CC integer-based DGHV scheme [20] to the RLWE-based BGV scheme [5], so that our scheme is able to handle the polynomial CRT packing. As in Section 3.3, an FHE scheme with packing generates a ciphertext in the polynomial CRT representation, which packs multiple plaintexts in the ciphertext slots.

To implement the packing method with Apriori, we choose to pack binary components column-by-column, where each column has N_{trans} components for all N_{items} columns and each ciphertext packs ℓ components of their components. That is, our scheme requires $\lceil N_{trans}/\ell \rceil$ ciphertexts to pack all N_{trans} components in one item-column as shown in Fig. 2 (b). Here, when N_{trans} is indivisible by ℓ, its remaining r components are packed into another ciphertext along with $\ell - r$ dummy zero components, as shown in Fig. 3.

With its polynomial CRT packing method, our scheme has two advantages. Firstly, the number of ciphertexts to represent all components in the database decreases from $N_{trans} \cdot N_{items}$ to $\lceil N_{trans}/\ell \rceil \cdot N_{items}$. Likewise, the costs of both memory/storage and communication between the client and the server decrease from $N_{trans} \cdot N_{items}$ to $\lceil N_{trans}/\ell \rceil \cdot N_{items}$, given that the space usage and communication cost arises from the ciphertext size. Secondly, the number of multiplications required to count the supports of all patterns decreases from $\Sigma_{i=1}^{m} p_i N_{trans}^{i-1}$ to $\Sigma_{i=1}^{m} p_i \lceil N_{trans}/\ell \rceil^{i-1}$, where p_i, N_{trans}^{i-1}, and m are defined above.

4.2 Optimization by Caching

We propose a caching technique to omit redundant operations when counting the support of each candidate in the Apriori algorithm with FHE. As described in Section 4.1, counting the support of each candidate requires $\Sigma_{i=1}^{m} p_i \lceil N_{trans}/\ell \rceil^{i-1}$ element-wise vector multiplications. In particular, with a length-$(i+1)$ candidate itemset $\boldsymbol{c}=\{c_1,c_2,\ldots,c_{i+1}\}$, the operation $\otimes_{j=1}^{i+1} c_j$is required for calculating its support, where $\otimes$ is the element-wise vector multiplication. For example, when we calculate the support of a length-4 candidate itemset $\{\boldsymbol{a,b,c,d}\}$, element-wise vector multiplications of $\boldsymbol{a}\otimes\boldsymbol{b}\otimes\boldsymbol{c}\otimes\boldsymbol{d}$ are required. However, the supports of the length-3 candidate itemsets (i.e., $\boldsymbol{a}\otimes\boldsymbol{b}\otimes\boldsymbol{c}$, $\boldsymbol{a}\otimes\boldsymbol{b}\otimes\boldsymbol{d}$, $\boldsymbol{a}\otimes\boldsymbol{c}\otimes\boldsymbol{d}$, and $\boldsymbol{b}\otimes\boldsymbol{c}\otimes\boldsymbol{d}$) have been calculated before for the length-4 one $\{\boldsymbol{a,b,c,d}\}$, because of the Apriori algorithm described in Section 3.1. To take the full advantage of this phenomena, we adopt a caching technique to reuse the previously calculated element-wise vector multiplication results.

During the execution of the $(i+1)^{th}$ iteration, we will reuse the cached result from the i^{th} iteration. In the example above, our algorithm caches the results $\boldsymbol{a}\otimes\boldsymbol{b}\otimes\boldsymbol{c}$, $\boldsymbol{a}\otimes\boldsymbol{b}\otimes\boldsymbol{d}$, $\boldsymbol{a}\otimes\boldsymbol{c}\otimes\boldsymbol{d}$, and $\boldsymbol{b}\otimes\boldsymbol{c}\otimes\boldsymbol{d}$ with indexation by items when counting the supports for length-3 patterns, and reuse them in the next iteration for length-4, as shown in Fig. 4 (b).

With the proposed caching technique, our algorithm requires only one time element-wise vector multiplication per support calculation in the i^{th} iteration. As

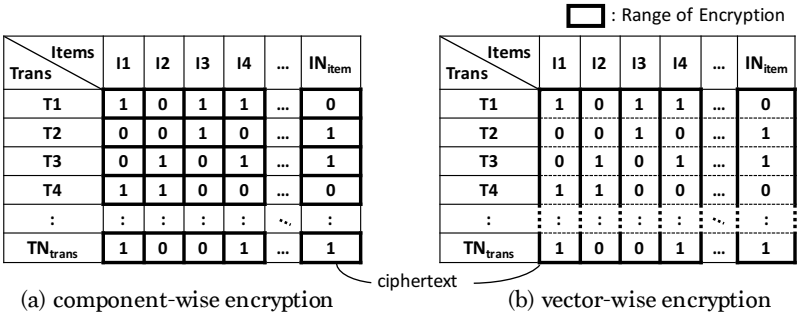


Fig. 2: Encryption strategy of item-transaction database

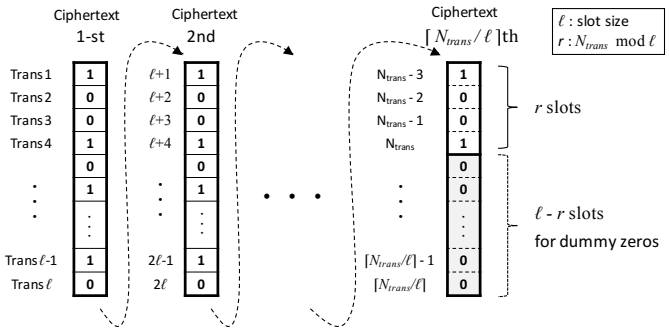
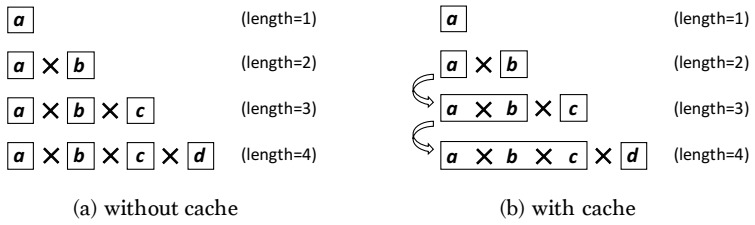


Fig. 3: Ciphertext-packing of item-column-components

described in Section 4.1, the total number of element-wise vector multiplications decreases from $\Sigma_{i=1}^{m} p_i \lceil N_{trans}/\ell \rceil^{i-1}$ without caching to $\Sigma_{i=1}^{m} p_i \lceil N_{trans}/\ell \rceil$ with it, where p_i and m are defined at the beginning of Section 4. The server's total computational order is equal to the computational order of the support counting, which decreases from $\mathcal{O}(\lceil N_{trans}/\ell \rceil^{i-1})$ to $\mathcal{O}(\lceil N_{trans}/\ell \rceil)$.

Our new algorithm for counting supports with caching technique is shown in Algorithm 3. The caching technique works when the pattern length is greater than two. In addition, we adopt the TotalSums function described in Section 3.4 to sum up all elements, i.e., slots of the CRT-represented ciphertext.



(a) without cache (b) with cache

Fig. 4: Support calculation with caching technique.

Algorithm 3 CountSupport by FHE with Cache $(ETDB, C, CD)$

Input: Encrypted transaction database $ETDB$; Candidate itemsets C;
Associative array for caching data CD;
Output: Support array of candidate itemsets S; Updated CD;

$S := \emptyset$;
for each candidate itemset $\boldsymbol{c} \in C$ **do**
  $i := \|\boldsymbol{c}\|$;
  $itemID := \boldsymbol{c}[i-1]$; # $i-1$: last element
  $\boldsymbol{c'} := \boldsymbol{c}[0, 1, \ldots, i-2]$; # length-$(i-1)$ itemset
  $\boldsymbol{col}$:= getItemColumnfromETDBbyID($itemID$);
  $hashKey$:= setHashKeyfromItemset($\boldsymbol{c'}$);
  $\boldsymbol{cache}$:= getCachedDatabyKey($CD, hashKey$);
  $\boldsymbol{res}$:= elementwiseVectorMultiply($\boldsymbol{cache}$, $\boldsymbol{col}$); # $\boldsymbol{cache} \cdot \boldsymbol{col}$
  $support$:= TotalSums($\boldsymbol{res}$); # sum up all elements of $\boldsymbol{res}$
  $newHashKey$:= makeNewHashKey($\boldsymbol{c}$);
  $cacheNewData$(makePair($newHashKey$, $\boldsymbol{res}$));
  S.append($support$);
end for
return S, CD;

5 Experimental Evaluation

In this section, we evaluate the effectiveness of both i) adopting the packing scheme implemented with Apriori, and ii) optimization by our caching technique. Furthermore, we confirm that our optimized scheme works acceptably with α-pattern uncertainty and relatively large data sizes.

The dataset we used for the experimental evaluations was one that was generated artificially by the IBM Quest Synthetic Data Generator[1]. This generator produces various patterns of datasets by changing the parameters $\{T, I, N, D, L\}$, where T is the average length of items per transaction, I is the average length of the maximal pattern, N is the number of different items in a transaction, D is the number of transactions, and L is the number of possible frequent patterns that can be generated.

In addition, our scheme is implemented both with the public FHE library HElib[2] that supports the BGV RLWE cryptosystem [5], and with the NTL mathematical library[3] over the GMP multiple-precision arithmetic library[4]. The GMP is used for the long integer arithmetic and the NTL is for handling polynomials over the integers. HElib builds an FHE scheme with parameters $\{p, r, k, l, c, w\}$, where p^r denotes the plaintext space, k is the security parameter, l is the number of levels in the modulus chain, c is the number of columns in the key-switching matrices, and w is the Hamming weight of the secret key.

The platform used in the evaluation consists of two machines: a client with an Intel Xeon CPU E5-2643 v3 running at 3.4 GHz and with 512 GB of memory, and a server with an Intel Xeon CPU E7-8880 v3 running at 2.3 GHz and with 1 TB of memory, both of which are equipped with CentOS6.6.

For comparison with the component-wise encryption scheme of P3CC, we implemented our method over the RLWE-based FHE, and then use it in the following evaluations. As for the HElib parameters, we set $\{p, r, k, l, c, w\}$ to $\{2, 14, 80, 10, 3, 64\}$ in the following evaluations. The plaintext space 2^{14} is higher than the largest value of D we use, the level $l = 10$ is to enable at least four multiplications per ciphertexts, and k, w, and c are default values for the security and the key-switching matrix.

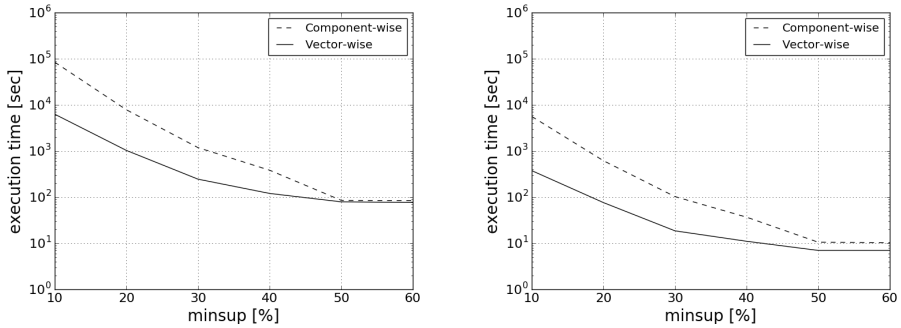
5.1 Experiment with Ciphertext Packing

To evaluate the effectiveness of our scheme with the ciphertext packing (i.e., a vector-wise encryption scheme), we compared it with our implementation of P3CC (i.e., a component-wise encryption scheme), from the viewpoints of both execution time and maximum memory usage. We choose the small dataset, T10I6N50D100L1k, for the comparative experiment, since the component-wise encryption scheme over the RLWE-based FHE takes a relatively long time to run.

[1] http://fimi.ua.ac.be/data/
[2] http://shaih.github.io/HElib/index.html
[3] http://www.shoup.net/ntl/
[4] https://gmplib.org/

Fig. 5 shows the execution times of both the component-wise and the vector-wise encryption schemes with variation of the minimum support. We first ran the experiment on a single thread (Fig. 5a), and then adopted parallelization by multi-thread file reading/writing, encryption, and calculation of each pattern's support (Fig. 5b). The client executed file writing and encryption on 12 threads, and the server executed file reading and calculations on 24 threads.

As shown in Fig. 5 (a), compared to the component-wise encryption scheme, our vector-wise one is 13.4 times faster with 10% minsup, and 7.26 times faster on average over the range $10\% \leq \text{minsup} \leq 40\%$. With parallelization, our scheme is 14.9 times faster with 10% minsup, and 7.97 times faster on average, as shown in Fig. 5 (b). The maximum memory usage decreases by 90.7%.



(a) on single thread (b) on multiple threads

Fig. 5: Comparison of packing and non-packing schemes

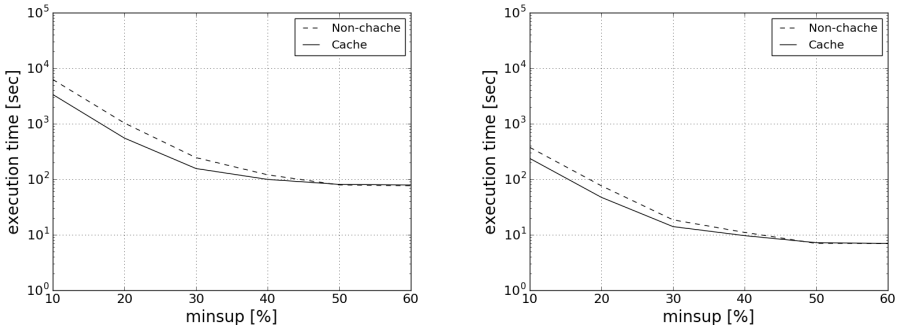
5.2 Optimization by Caching

To evaluate our optimized scheme with both ciphertext packing and the caching technique, we compare it with the non-optimized scheme, i.e., only with ciphertext packing, which was evaluated in Section 5.1. We used the same dataset to compare them for the same criteria.

Fig. 6 shows the execution times of both the optimized and non-optimized schemes with variation of the minimum support. We ran these on a single thread (Fig 6a), and then on multiple threads (Fig. 6b). The number of threads and the target of multi-threading were the same as those of the evaluations in Section 5.1.

As shown in Fig. 6 (a), compared to the non-cached scheme, our scheme is 1.86 times faster with 10% minsup, and 1.62 times faster on average over the range $10\% \leq \text{minsup} \leq 40\%$. With parallelization, our scheme is 1.58 times faster

with 10% minsup, and 1.42 times faster on average, as shown in Fig. 6 (b). The maximum memory usage decreases by 14.5%. In total, our optimized scheme is 23.6 times faster, and the memory usage decreases by 92.1% with 10% minsup in comparison with the multi-threaded component-wise encryption scheme in Section 5.1.



(a) on single thread (b) on multiple threads

Fig. 6: Comparison of caching and non-caching schemes

5.3 Security and Data Size

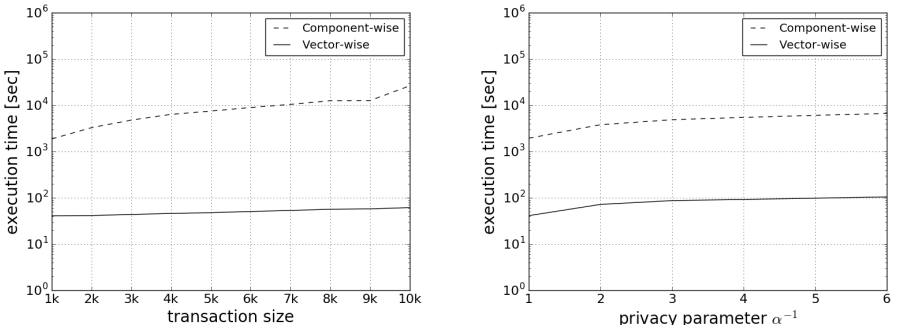
We test the scalability of our optimized scheme (i.e., with packing and caching) for i) the size of the data, and ii) α-pattern uncertainty security as described in Section 3.2. The latter determines the server's uncertainty over true patterns in the candidate itemset by adding dummy patterns.

We evaluate the first case by measuring the execution times and memory usages of both our optimized scheme and the component-wise encryption scheme, while varying the transaction data size. We first set the dataset to T10I6N50D1kL1k, and then vary the parameter D from 1k to 10k as shown in Fig. 7 (a). The experiment is multi-threaded with the same conditions as in Section 5.1, with 20% minsup.

As shown in Fig. 7 (a), the difference in the execution times between the schemes increases with the transaction size. This result is attributed to the number of ciphertexts generated in each scheme by the ciphertext packing, as described in Section 4.1. Compared to the component-wise encryption scheme, our scheme is 430 times faster with $D = 10$k, and 180 times faster on average over the range $1\text{k} \leq D \leq 10\text{k}$. In addition, the maximum memory usage of our scheme decreases by 94.7%.

We then evaluate the second case by measuring both the execution time and the memory usage of our optimized scheme, with α-pattern uncertainty as

described in Section 3.2. The parameter α is the probability of inferring the true patterns in the candidate set with dummy patterns. In other words, if as α increases, the security becomes weaker. Therefore, we can consider α^{-1} to be a privacy parameter. We first set the dataset to T10I6N50D1kL1k and α^{-1} to 1 (i.e., no dummy patterns), then vary the parameter α^{-1} from 1 to 6 as shown in Fig. 7 (b). Compared to the component-wise encryption scheme, our scheme is 56.8 times faster on average over the range $1 \leq \alpha^{-1} \leq 6$. In addition, the maximum memory usage decreases by 82.8%. There is a trade-off between the execution time and the security due to the additional calculations for the dummy patterns.



(a) varying no. of transactions (b) varying privacy parameter

Fig. 7: Optimized scheme with security and relatively large data sizes

6 Conclusions and Future Work

We proposed an efficient and secure frequent pattern mining by adopting both the polynomial CRT packing method and a caching technique. Our experimental results shows that the proposed scheme has lower time and space complexities in comparison with those of the previous P3CC scheme. When the transaction size is 10,000, our optimized scheme is 430 times faster and the memory usage decreases by 94.7%.

Future work will include attempting to reduce the communication costs by comparing larger and smaller ciphertexts. To achieve this, bootstrapping procedures will have to be implemented, since such comparisons require numerous homomorphic operations. Moreover, we will consider a new security idea that should work for the aforementioned comparative scenario.

Acknowledgements. This work was supported by the CREST program of the Japan Science and Technology Agency. We would like to thank Mr. Takumi Takahashi, who implemented our scheme experimentally.

References

1. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: ACM SIGMOD record. vol. 22, pp. 207–216. ACM (1993)
2. Agrawal, R., Srikant, R., et al.: Fast algorithms for mining association rules. In: Proceedings of the 20th international conference on very large data bases, VLDB. vol. 1215, pp. 487–499 (1994)
3. Atzori, M., Bonchi, F., Giannotti, F., Pedreschi, D.: Anonymity preserving pattern discovery. vol. 17, pp. 703–727. Springer-Verlag New York, Inc. (2008)
4. Bhaskar, R., Laxman, S., Smith, A., Thakurta, A.: Discovering frequent patterns in sensitive data. In: Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 503–512. ACM (2010)
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325. ACM (2012)
6. Evfimievski, A., Gehrke, J., Srikant, R.: Limiting privacy breaches in privacy preserving data mining. In: Proceedings of the 22th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 211–222. ACM (2003)
7. Gellman, R.: Privacy in the clouds: risks to privacy and confidentiality from cloud computing. In: Proceedings of the World Privacy Forum, 23 February (2012)
8. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41th Annual ACM Symposium on Theory of Computing. pp. 169–178. ACM (2009)
9. Graepel, T., Lauter, K., Naehrig, M.: Ml confidential: Machine learning on encrypted data. In: Information Security and Cryptology–ICISC 2012, pp. 1–21. Springer (2012)
10. Halevi, S., Shoup, V.: Algorithms in helib. In: International Cryptology Conference, pp. 554–571. Springer (2014)
11. Kaosar, M.G., Paulet, R., Yi, X.: Fully homomorphic encryption based two-party association rule mining. vol. 76, pp. 1–15. Elsevier (2012)
12. Kapoor, V., Poncelet, P., Trousset, F., Teisseire, M.: Privacy preserving sequential pattern mining in distributed databases. In: Proceedings of the 15th ACM international conference on Information and knowledge management. pp. 758–767. ACM (2006)
13. Khedr, A., Gulak, G., Vaikuntanathan, V.: Shield: Scalable homomorphic implementation of encrypted data-classifiers. In: IEEE Transactions on Computers. IEEE (2015)
14. Liu, J., Li, J., Xu, S., Fung, B.C.: Secure outsourced frequent pattern mining by fully homomorphic encryption. In: Big Data Analytics and Knowledge Discovery, pp. 70–81. Springer (2015)
15. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop. pp. 113–124. ACM (2011)

16. Qiu, L., Li, Y., Wu, X.: Protecting business intelligence and customer privacy while outsourcing data mining tasks. vol. 17, pp. 99–120. Springer (2008)
17. Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Public Key Cryptography–PKC 2010, pp. 420–443. Springer (2010)
18. Smart, N.P., Vercauteren, F.: Fully homomorphic simd operations. vol. 71, pp. 57–81. Springer (2014)
19. Tai, C.H., Yu, P.S., Chen, M.S.: k-support anonymity based on pseudo taxonomy for outsourcing of frequent itemset mining. In: Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 473–482. ACM (2010)
20. Van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Advances in cryptology–EUROCRYPT 2010, pp. 24–43. Springer (2010)
21. Wang, Y., Wu, X.: Approximate inverse frequent itemset mining: Privacy, complexity, and approximation. In: 5th IEEE International Conference on Data Mining (ICDM). pp. 482–489. IEEE (2005)