Privacy Preserving Data Mining of Sequential Patterns for Network Traffic Data

Seung-Woo Kim[1], Sanghyun Park*[1], Jung-Im Won[2], and Sang-Wook Kim[2]

[1] Department of Computer Science
Yonsei University, Korea
{kimsw,sanghyun}@cs.yonsei.ac.kr
[2] College of Information and Communications
Hanyang University, Korea
{jiwon,wook}@hanyang.ac.kr

Abstract. As a total amount of traffic data in networks has been growing at an alarming rate, many researches to mine traffic data with the purpose of getting useful information are currently being performed. However, since network traffic data contain the information about Internet usage patterns of users, network users' privacy can be compromised during the mining process. In this paper, we propose an efficient and practical method for privacy preserving sequential pattern mining on network traffic data. In order to discover frequent sequential patterns without violating privacy, our method uses the *N-repository server model* that operates as a single mining server and the *retention replacement technique* that changes the answer to a query probabilistically. In addition, our method accelerates the overall mining process by maintaining the *meta tables* in each site. Extensive experiments with real-world network traffic data revealed the correctness and the efficiency of the proposed method.

Keywords: Data mining, Sequential pattern, Network traffic, Privacy

1 Introduction

Owing to the rapid advance of network technology, the number of computers connected to the Internet increases dramatically, so does the information delivered over the vast Internet. Recently, there has appeared a new kind of data mining researches that extract useful knowledge from network traffic data automatically gathered by a remote server [9, 7, 11].

Table 1 shows an example of network traffic data gathered by Ethereal[3]. The network traffic data have the following characteristics: First, there exist various

* Corresponding Author
[3] http://www.ethereal.com/

timestamp	source address	source port	destination address	destination port
13:37:11.950966	180.1.1.1	36872	amazon.com	www
13:37:11.954474	amazon.com	www	180.1.1.1	36872
13:37:22.384472	180.1.1.1	36915	192.168.1.3	telnet
13:37:22.385327	192.168.1.3	telnet	180.1.1.1	36915

Table 1. An example of network traffic data gathered by Ethereal.

kinds of data since all the computers connected to the Internet can produce network traffic data potentially. Second, a huge amount of network traffic data are accumulated due to frequent actions for data sending/receiving by a lot of computers. Third, network traffic data are scattered over a large number of sites.

Sequential pattern mining is the most useful for this application since the order of events has an important meaning in network traffic data [9, 7].

Network traffic data contain detailed information of Internet usage for every user, which informs that a user accesses a site at a time specifically. Herein, data mining on network traffic data has the problem of compromising privacy of network users. Therefore, it requires sophisticated techniques for hiding or reforming users' private information during a data gathering process. Moreover, these techniques should not sacrifice the correctness of mining results.

Privacy preserving data mining is a new kind of a research area that aims at mining data with guaranteeing privacy of individual users [4, 13, 2, 5, 8, 6, 10, 12, 14]. Recently, there have been many research efforts performed in this area. Most methods proposed in prior studies, however, manage data in a few sites or deal with a small number of distinct types of data. Thus, these methods are not appropriate for mining network traffic data since they suffer from the problems of incorrectness and low performance.

In this paper, we discuss solutions to the problems that occur in previous methods. We propose a novel method for sequential pattern mining on network traffic data. The proposed method preserves privacy of sites and guarantees the correctness of mining results. The method discovers frequently-occurring network traffic patterns with hiding site information through two ways: (1) It employs the *N-repository server model* that makes multiple servers behave as a single mining server; (2) It uses the *retention replacement* technique that changes the answer by a given probability. Also, the method maintains *meta tables* in each site so as to quickly determine whether candidate patterns ever occurred in the site, thereby making the overall mining process become highly efficient.

2 Related Work

Clifton et al. [4] firstly raised the privacy problem in data mining and motivated subsequent studies [13, 2, 5, 8, 6, 10, 12, 14] that aimed to solve the problem.

In the method proposed in [2], in order to preserve privacy, each site changes the original numeric value of an individual item before sending the value to the

server by adding an arbitrary value selected from given probability distribution. The server builds a decision tree by reconstructing actual value distribution.

Other method called the *retention replacement* [13, 3] perturbs and reconstructs data in gathering and mining, respectively, for privacy preservation. For every data whose element represents 0 or 1, each site sends the original value with probability p and the perturbed one with probability $(1-p)$. For gathered data, the server counts the total numbers of 1's and 0's, and then estimates the original numbers of 1's and 0's. This method is applicable only to boolean data.

Some later studies [5, 3] tried to apply those two methods to various data types, however, showed lower accuracy as the number of data types increases.

The method proposed by Rizvi et al. [13] uses the *retention replacement* for finding frequent itemsets. This can be applied to the case where item types to occur are pre-determined. Considering network traffic data where a large number of item types occur, we can hardly determine all the item types in advance. Also, this method finds frequent patterns via a whole database scan and thus is very inefficient since network traffic data are very huge.

The method proposed in [8] collects local frequent itemsets from sites by employing a commutative encryption and obtains global frequencies of itemsets by employing a secure sum which uses a random number. For performing a commutative encryption and a secure sum, this method has to serially send data in the cycle of sites. This requires a lot of time in case of a large number of sites. Fukasawa et al. [6] improved the efficiency and security of this method. However the improved one still has cycling communications.

Zhan et al. proposed a method for sequential pattern mining with privacy preservation [14]. This method mainly targets a distributed environment where vertical partitioning without duplication is employed. In our situation, duplicated data could occur in more than one site since multiple PCs can access the same Internet site. Therefore, this method is inapplicable to network traffic data.

In the method proposed in [10], a secure protocol is used for mining a decision tree classifier from distributed sites. Pinkas [12] showed how protocols for secure distributed computation can be employed for privacy preservation, however he also pointed out that the performance of protocols should be improved.

In summary, prior studies have the problems applying to a large amount of network traffic data. First, due to a variety of data types, previous methods are not directly applicable and cannot obtain accurate mining results. Second, since there exist a large number of sites and data can be duplicated, previous methods targeted for a distributed database environment have limitations on practicality.

3 Problem Definition

Network traffic data are normally gathered by a tcp/ip data capture program such as Ethereal. In this paper, we aim at finding sequential patterns from network traffic data without disclosing data in each site. First, we simplify the network traffic data in the form of Table 1 as those in the form of Table 2. In Table 2, "out" denotes sending and "in" does receiving.

timestamp	in/out	address	timestamp	in/out	address
13:37:11.950966	out	amazon.com	13:37:22.384472	out	192.168.1.3
13:37:11.954474	in	amazon.com	13:37:22.385327	in	192.168.1.3

Table 2. An example of network traffic data reconstructed.

In order to find temporal relationship among events in network traffic data, we can apply sequential pattern mining methods [9, 7]. We impose a restriction that two adjacent items in network traffic data should have a time interval smaller than or equal to a predefined $MaxGap$ value to be regarded as related.

We formulate the problem we are going to solve as follows: Given t sites T_1, T_2, ... , T_t, the maximum time interval $MaxGap$, and the minimum support $MinSup$, we discover all the sequential patterns, which have a support larger than $MinSup$ and a time interval between any pair of adjacent items equal to or smaller than $MaxGap$. We assume that a site stores network traffic data as in the form of Table 2.

A mining process should also satisfy the condition for preserving privacy in every site. Let us denote a set of sites, where network traffic has occurred, as E and a set of network traffic data as I. In a mining process, an element e_j in E is opened since it participates in the mining process; Also, an element i_k in I is also opened since it should be contained in a result of mining. However, a pair of (e_j, i_k), which says a site e_j has been connected to an IP address i_k, should not be opened in a mining process. We define this condition for preserving privacy.

4 Proposed Method

4.1 Overall mining process

F_1	A set of all frequent sequential patterns of length 1 (or large 1-sequences, frequent items)
F_k	A set of all frequent sequential patterns of length k (or large k-sequences)
C_k	A set of all candidate patterns of length k

Table 3. Definition of symbols.

The proposed mining process consists of four phases. Table 3 shows the definition of symbols which are used to explain the mining process. The first phase utilizes the *N-repository server model* to safely discover F_1. The second phase generates C_{k+1} by self-joining F_k. k is initialized to 1 when the second phase is executed for the first time. If C_{k+1} is empty, we enter into the final phase. Otherwise, we enter into the third phase. For each candidate in C_{k+1}, the third

phase sends every site the query asking whether the candidate has ever occurred in the site. After receiving the answers from all sites, the third phase judges whether each candidate is frequent or not, and then constructs F_{k+1}, with the candidates judged as frequent. The third phase then increases k by 1 and calls the second phase. The final phase prints all frequent patterns in F_1, F_2, ... , and F_k, and stops the mining process.

4.2 Finding frequent items using *N-repository server model*

The proposed *N-repository server model* finds F_1, without compromising the condition for preserving privacy by concealing the linkage between the site identifier and the traffic data, (e_j, i_k). More specifically, it obscures their linkage by encrypting the traffic data, i_k, at the first step and by aggregating the site identifiers, e_j, at the second step.

The proposed *N-repository server model* consists of N servers, $\{S_1, S_2, \ldots, S_N\}$, and N pairs of encryption keys and decryption keys, $\{(EK_1, DK_1), (EK_2, DK_2), \ldots, (EK_N, DK_N)\}$. Each site has all encryption keys but server S_i has only a decryption key DK_i $(1 \leq i \leq N)$. To find frequent items safely, the *N-repository server model* operates as follows:

1. Each site classifies the items (i.e., the traffic data) into N groups, $\{G_1, G_2, \ldots, G_N\}$, using a hash function.
2. Each site encrypts the items in G_i with encryption key EK_i $(1 \leq i \leq N)$.
3. Each site sends the encrypted items in G_i to server S_{i+1} $(1 \leq i \leq N-1)$ and the encrypted items in G_N to server S_1.
4. Each server performs the aggregation on the encrypted items to obtain the number of occurrences of each encrypted item and then picks up the encrypted *frequent* items.
5. Each server S_i sends encrypted *frequent* items to server S_{i-1} $(2 \leq i \leq N)$ and server S_1 does to server S_N.
6. Each server S_i decrypts the received items with its decryption key DK_i and then reports the frequent items to public.

We assume that the servers in our model operate in a *semi-trusted* operation model. In the *semi-trusted* operation model, servers may try to acquire private data but do not cooperate with other servers to do that. This semi-trusted operation model is common in real environments where one wants to get the result of computation but is not willing to offer one's own data to others [14].

4.3 Finding frequent patterns longer than one

After finding out F_1, we have to sequentially discover frequent patterns longer than one. At first, one of N servers is elected as a principal mining server. To discover all frequent patterns longer than one, the principal mining server assigns 1 to variable k and executes the following steps.

1. It produces C_{k+1} by self-joining F_k in the same way as *Apriori* algorithm [1]. It executes step 5 if C_{k+1} is empty. Otherwise, it executes step 2.
2. For each candidate pattern CP in C_{k+1}, the server sends every site T a query asking whether CP has ever occurred in T or not.
3. Each site T sequentially inspects traffic data or the *meta tables*, which will be described in Section 4.4, to determine CP's occurrence in T. An actual answer of the query would be 1 or 0 as CP's occurrence. However, to preserve the privacy of the site, the actual answer is perturbed by the *retention replacement* [13, 3].
4. For each query, the principal mining server aggregates the counts of the sites answered 1 and the sites answered 0. Then, using the two counts, it conjectures the number of sites whose actual answers were 1. It then compares that number with $MinSup$ and constructs F_{k+1}, by choosing from C_{k+1}. It finally increases k by 1 and calls step 1.
5. When it reaches this step, C_{k+1} is empty. Therefore, it prints all the frequent patterns discovered and then stops the execution of the algorithm.

4.4 Meta tables to quickly determine the occurrence or non-occurrence of candidate patterns

In the *Apriori* algorithm, patterns of length k can be regarded as candidate patterns only when all of their sub-patterns are frequent. However, in the sequential pattern mining with time constraints, even the patterns containing infrequent sub-patterns can be treated as candidate patterns if all of their sub-patterns occurring *contiguously* in the underlying patterns are frequent. Therefore, compared to the mining techniques based on the original *Apriori* algorithm, the sequential pattern mining with time constraints impose less requirement for patterns to be treated as candidate patterns. As a result, more candidate patterns are generated and, to accelerate the overall mining process, it is crucial to handle each candidate pattern efficiently.

In this paper, we employs special-purpose *meta tables* in each site T for speeding-up the process to decide the occurrence or non-occurrence of CP in T.

Meta tables for storing pairs of items satisfying *MaxGap*

Let m denote the number of frequent items. At first, the principal mining server sends out the list of all frequent items to each site T. Then, site T lexicographically sorts the frequent items and assigns each frequent item the corresponding lexicographic order. Site T then stores the name and lexicographic order of each frequent item into the meta table called FreqItems. FreqItems consists of two columns, ItemName and Order. Given a frequent item, ItemName and Order store its name and lexicographic order, respectively.

The second meta table is OccTs_OccBits. This table consists of three columns, Order, OccTs, and OccBits. For each frequent item FI in the traffic data of T, Order stores the lexicographic order of FI, and OccTs stores the timestamp at which FI occurred, and OccBits stores a bit-vector of length m whose i^{th} bit

indicates whether or not the frequent item of lexicographic order i has ever occurred within $MaxGap$ after the occurrence of FI. We denote the i^{th} bit of OccBits as OccBits(i). OccTs_OccBits can be constructed by scanning the entire traffic data in site T. The number of tuples in OccTs_OccBits is same as the number of occurrences of frequent items in T.

The third meta table is OccCnts. OccCnts consists of $m+1$ columns, Order, Cnt_1, Cnt_2, ... , Cnt_m. OccCnts has a tuple for each frequent item and therefore contains m tuples. Let us consider the i^{th} tuple of OccCnts. It has i as a value of Order. As a value of Cnt_j, it has the number of occurrences of the frequent item of order j whose timestamps are within $MaxGap$ after the occurrences of the frequent item of order i. The i^{th} tuple of OccCnts can be populated by queryingOccTs_OccBits.

An example of *meta tables* maintained within a single site is shown in Fig. 1.

< FreqItems >

ItemName	Order
A	1
B	2
C	3

< OccTs_OccBits >

Order	OccTs	OccBits	Order	OccTs	OccBits
1	13:37:11.95	000	2	13:38:17.23	010
1	13:37:32.43	010	2	13:38:19.12	000
1	13:38:07.05	001	3	13:37:35.17	000
1	13:38:15.51	010	3	13:38:08.54	000
2	13:37:34.21	001	3	13:38:12.31	001
2	13:38:05.44	100	3	13:38:14.08	100

< OccCnts >

Order	Cnt_1	Cnt_2	Cnt_3
1	0	2	1
2	1	1	1
3	1	0	1

Fig. 1. An example of *meta tables* maintained within a single site.

Determining the occurrences or non-occurrences of candidate patterns

The algorithm to determine the occurrences or non-occurrences of candidate patterns using the *meta tables* has the following 4 steps.

1. Divide a candidate pattern
   Let $CP_n = \langle I_1, I_2, ..., I_n \rangle$ denote a candidate pattern with n items. We first divide CP_n into $n-1$ sub-patterns each of which consists of two adjacent items of CP_n. The j^{th} sub-pattern of CP_n is denoted as $SP_j = \langle I_j, I_{j+1} \rangle$ $(j = 1, 2, ..., n-1)$.

2. Determine the execution orders of the sub-patterns
   Sub-patterns are executed on the *meta tables* and their results are joined with those of other sub-patterns. If we are able to discover the optimal execution orders which minimize the sizes of intermediate results, then we can determine the occurrence or non-occurrence of a candidate pattern as early as possible. The simplest way to find out the optimal execution orders is to consider all the possible combinations of execution orders and to choose the one which will produce the smallest intermediate results. However, there are $(n-1)!$ distinct combinations for $n-1$ sub-patterns and thus such a

simple approach is not scalable to large n. Therefore, we employ the following *greedy* algorithm which quickly discovers near-optimal execution orders.

(a) We choose the sub-pattern which will have the smallest result set size, and let 1 be its execution order. We then assign 1 to variable k.
(b) Let us assume that the execution order of SP_j has just been decided as k. To decide the sub-pattern of execution order $k+1$, we decrease j' from $j-1$ to 1 until we find $SP_{j'}$ whose execution order is not decided yet. Also, we increase j'' from $j+1$ to $n-1$ until we find $SP_{j''}$ whose execution order is not decided yet.
(c) If neither $SP_{j'}$ nor $SP_{j''}$ exists, we stop the execution of the *greedy* algorithm. However, if $SP_{j''}$ does not exist but $SP_{j'}$ exists, then we let $k+1$ be the execution order of $SP_{j'}$. On the contrary, if $SP_{j'}$ does not exist but $SP_{j''}$ exists, then we let $k+1$ be the execution order of $SP_{j''}$. If both $SP_{j'}$ and $SP_{j''}$ exist, then we choose the one which will have a smaller result set size and let $k+1$ be its execution order. If their result set sizes will be same, then we choose the one farther from the corresponding end. This reduces the possibility of the absence of either $SP_{j'}$ or $SP_{j''}$ in the next step and therefore enables to obtain a better combination of execution orders.
(d) We increase k by one and return to step 2(b).

In the middle of this *greedy* algorithm, there is a step to calculate the result set sizes of sub-patterns. The result set size of $SP_j = \langle I_j, I_{j+1} \rangle$ is equal to the number of occurrences of item I_{j+1} within $MaxGap$ after the occurrences of item I_j. The result set size of a sub-pattern can be easily obtained by using two *meta tables*, FreqItems and OccCnts, who were explained above.

3. Execute the sub-patterns and join their results

According to the execution orders obtained in step 2, we execute all sub-patterns one by one while joining their intermediate results. That is, for each k from 1 to $n-1$, we execute the following steps.

(a) For the two items of the sub-pattern whose execution order is k, we find their lexicographic orders using FreqItems. Let p and q be the lexicographic orders of the preceding item and the succeeding item, respectively.
(b) We execute the following SQL statement to obtain the result set RS_k of the sub-pattern of execution order k.

```
select p, OccTs, q      // p and q are not column names but constants
into RS_k
from OccTs_OccBits
where Order = p and OccBits(q) = 1;
```

(c) We join the result set RS_k with JRS_{k-1}, the intermediate result set obtained by sequentially joining all the result sets of sub-patterns of execution orders from 1 to $k-1$, producing a new intermediate result set JRS_k. For a simpler explanation, let us rename the tables to be joined as follows. If the sub-pattern of execution order k is on the left

of the sub-patterns of execution orders from 1 to $k-1$, then we rename the sub-pattern of execution order k as TA and the intermediate result set JRS_{k-1} as TB. Otherwise, we rename the sub-pattern of execution order k as TB and the intermediate result set JRS_{k-1} as TA. Then, the conditions for a tuple ta of TA to be joined with a tuple tb of TB are like the following:
  - **Join condition 1:** The last item of ta must be identical to the first item of tb.
  - **Join condition 2:** The gap from the timestamp of tb's first item to the timestamp of ta's last item must not be larger than $MaxGap$.

(d) We check if the join result JRS_k is empty. If so, we proceed to step 4. Otherwise, we increase k by one and return to step 3(a).

4. Determine the occurrence or non-occurrence of a candidate pattern
   We check if the final result set of step 3 is empty. If so, we conclude that the candidate pattern being considered has never occurred in this site. Otherwise, we judge that there is at least one occurrence of the candidate pattern.

Meta table to quickly judge the non-occurrence of a candidate pattern

Apriori algorithm joins frequent patterns of length n with themselves to generate candidate patterns of length $n+1$.

Let $\{CP_n\}$ denote the set of candidate patterns of length n. Also, let $\{CP'_n\}$ denote the set of candidate patterns in $\{CP_n\}$ whose occurrences were detected in the site. Now, let us consider a candidate pattern of length $n+1$, CP_{n+1}, delivered to the site most recently. If we break CP_{n+1} into two sub-patterns of length n, $CP_{n+1}[1..n]$ and $CP_{n+1}[2..n+1]$, then both of them are certainly elements of $\{CP_n\}$. The prerequisites for CP_{n+1} to occur in the site are the occurrences of both $CP_{n+1}[1..n]$ and $CP_{n+1}[2..n+1]$. Therefore, if either $CP_{n+1}[1..n] \notin \{CP'_n\}$ or $CP_{n+1}[2..n+1] \notin \{CP'_n\}$ is satisfied, then we can quickly recognize the non-occurrence of CP_{n+1} without looking up the *meta tables*. We implement this pruning method by a meta table named OccCandPatt, which stores occurred candidate patterns, in each site.

This pruning method enables to quickly judge the non-occurrence of a candidate pattern but increases the size of OccCandPatt continually. However, note that this method requires only the candidate patterns of length n in order to determine the non-occurrence of a candidate pattern of length $n+1$. Therefore, when the site receives from the principal mining server a candidate pattern of length $n+1$ for the first time, it removes the candidate patterns of length $n-1$.

5 Performance evaluation

5.1 Environment for experiments

In experiments, we collected 5,024,295 traffic data by Ethereal during 5 days. From them, we extracted 747,000 traffic data related to 736 IP addresses. The average inter-arrival time is 462.38 msec.

We compared the performances of three methods: Naive, OccTs, and OccTs+OccCandPatt. In order to discover F_1, Naive uses the *retention replacement* for all traffic data. Furthermore, it scans the original traffic data to verify whether every candidate is actually frequent. OccTs discovers F_1 by using the *N-repository server model* and F_k ($k \geq 2$) by the *retention replacement*. OccTs decides whether a candidate pattern is frequent by searching *meta tables* OccTs_OccBits and OccCnts. Finally, OccTs+OccCandPatt, which is basically based on OccTs, additionally uses meta table OccCandPatt. Furthermore, both OccTs and OccTs+OccCandPatt employ a *greedy* algorithm to determine the execution order of sub-patterns.

As a measure for evaluating accuracy, we used *Recall* and *Precision*. *Recall* indicates how much fraction are mined from all the actually frequent ones. *Precision* indicates how much fraction of mined patterns are actually frequent. As a performance measure, we used the average elapsed times in mining sequential patterns of the maximum length 6.

The hardware platform is the Pentium IV 3.0GHz PC equipped with 512 Mbytes main memory and 80 Gbytes hard disk of 7200RPM. The software platform is the Windows XP and the Java 2 Runtime Environment 1.4.2.

Since we got quite good performances in our parameter setting experiments, we set $MinSup$,$MaxGap$, the number of sites, and the number of servers to 0.2, 20, 10, and 5, respectively in the following experiments.

5.2 Analysis of accuracy

In order to evaluate accuracy of the proposed *N-repository server model*, we compared *Recall* and *Precision* of OccTs and Naive. Because the accuracy of both OccTs and OccTs+OccCandPatt is the same, we show only Naive and OccTs.

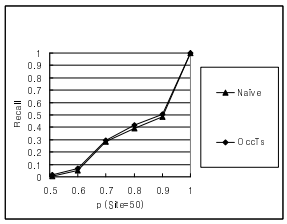
We evaluated *Recall* and *Precision* with different probability p. We set the number of sites to 50. Fig. 2 shows the results with p of 0.51 to 1. We note that the *retention replacement* is inapplicable with p of 0.5 [13].

The results show that, in Naive and OccTs, both *Recall* and *Precision* get higher as p gets close to 1. This is due to the *retention replacement* used in both methods to find frequent sequential patterns whose length is longer than 1. OccTs performs 1.04 to 1.20 and 1.01 to 1.12 times better than Naive in *Recall* and *Precision*, respectively.

Naive is inapplicable to analyzing real Internet traffic data because it has to know all the items likely to occur in advance. Furthermore, in the above two experiments, OccTs showed accuracy higher than Naive.

5.3 Analysis of performance

In order to evaluate the performance of OccTs and OccTs+OccCandPatt, we compared them with Naive in terms of the elapsed time for mining. We measured the elapsed time with different numbers of traffic data in each site. We set probability p in *retention replacement* to 1 in order to evaluate the average elapsed times of all the methods fairly. Fig. 3 shows the result.



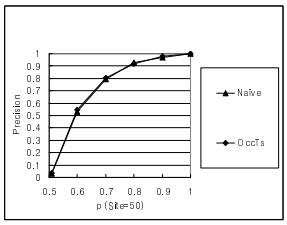


Fig. 2. *Recall* and *Precision* with different probability p.

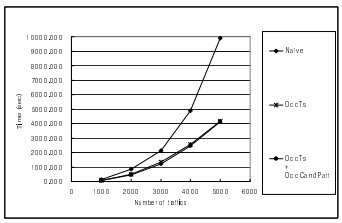


Fig. 3. The elapsed time with different numbers of traffic data.

We observe that, in all three methods, as the volume of traffic data gets larger, the elapsed time increases. This is because more frequent patterns appear with a larger volume of traffic data. OccTs performed 1.60 to 2.38 times better than Naive. It stores all pairs of frequent items that occur within $MaxGap$ into a meta table OccTs_OccBits and quickly determines whether candidate patterns occur by joining these pairs without accessing the network traffic data.

OccTs+OccCandPatt ran 1.01 to 1.10 times faster than OccTs. By referring to OccCandPatt, it examines whether candidate patterns have ever occurred in the site before searching OccTs_OccBits. Therefore, it achieves the pruning effect in the mining process. That is, the total elapsed time decreases because the number of candidate patterns to be searched in OccTs_OccBits gets smaller.

6 Concluding Remarks

In this paper, we have proposed a practical method for sequential pattern mining on network traffic data. The proposed method preserves privacy of sites and provides high accuracy of mining results. The proposed method can be used for finding frequent sequential visiting patterns of web pages. The mining results can be applied to prefetching of web pages and load balancing in web servers.

The contributions of the paper are summarized as follows: First, we have proposed a privacy preserving method that mines frequent sequential patterns from network traffic data. Our method uses the *N-repository server model* that operates as a single mining server and also employs the *retention replacement technique* that changes the answer by a given probability. Second, we have designed *meta tables* maintained in each site so as to quickly determine whether candidate patterns ever occurred in the site. Third, we have demonstrated the correctness and the efficiency of the proposed method via extensive experimentation with real-world network traffic data.

Acknowledgements

This work was supported by the Seoul R&BD Program(10561) in 2006, Korea Research Foundation Grant funded by Korea Government (MOEHRD, Basic

Research Promotion Fund) (KRF-2005-206-D00015), and the MIC of Korea under the ITRC support program supervised by the IITA (IITA-2005-C1090-0502-0009).

References

1. R. Agrawal and R. Srikant, "Mining Sequential Patterns," In Proceedings of the 11th IEEE International Conference on Data Engineering, pp. 3–14, 1995.
2. R. Agrawal and R. Srikant, "Privacy-Preserving Data Mining," In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 439–450, 2000.
3. R. Agrawal, R. Srikant, and D. Thomas, "Privacy Preserving OLAP," In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 251–262, 2005.
4. C. Clifton and D. Marks, "Security and Privacy Implication of Data Mining," In Proceedings of the 1996 ACM Workshop on Data Mining and Knowledge Discovery, pp. 15–19, 1996.
5. A. Evfimievski, R. Srikant, R. Agrawal, and J. Gehrke, "Privacy Preserving Mining of Association Rules," In Proceedings of the 2002 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 217–228, 2002.
6. T. Fukasawa, J. Wang, T. Takata, and M. Miyazaki, "An Effective Distributed Privacy-Preserving Data Mining Algorithm," In Proceedings of the 5th International Conference on Intelligent Data Engineering and Automated Learning, pp. 320–325, 2004.
7. Y. Hu and B. Panda, "A Data Mining Approach for Database Intrusion Detection," In Proceedings of the 2004 ACM Symposium on Applied Computing, pp. 711–716, 2004.
8. M. Kantarcioglu and C. Clifton, "Privacy-Preserving Distributed Mining of Association Rules on Horizontally Partitioned Data," In Proceedings of the 2002 ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, pp. 24–31, 2002.
9. W. Lee, S. Stolfo, and K. Mok, "A Data Mining Framework for Building Intrusion Detection Models," In Proceedings of IEEE Symposium on Security and Privacy, pp. 120–132, 1999.
10. Y. Lindell and B. Pinkas, "Privacy Preserving Data Mining," In Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology, pp. 36–54, 2000.
11. J. Luo and S. Bridges, "Mining Fuzzy Association Rules and Fuzzy Frequency Episodes for Intrusion Detection," International Journal of Intelligent Systems, Vol. 15, No. 8, pp. 687–704, 2000.
12. B. Pinkas, "Cryptographic techniques for privacy-preserving data mining," ACM SIGKDD Explorations Newsletter, Vol. 4, No. 2, pp. 12–15, 2002.
13. S. Rizvi and J. Haritsa, "Maintaining Data Privacy in Association Rule Mining," In Proceedings of the 28th International Conference on Very Large Data Bases, pp. 682–693, 2002.
14. J. Zhan, L. Chang, and S. Matwin, "Privacy-Preserving Collaborative Sequential Pattern Mining," In Proceedings of SIAM International Workshop on Link Analysis, Counter-terrorism, and Privacy, pp. 61–72, 2004.