CYBERNETICA
Institute of Information Security

Privacy-preserving Histogram Computation and Frequent Itemset Mining with Sharemind

Dan Bogdanov, Roman Jagomägis, Sven Laur

T-4-8 / 2009

Dan Bogdanov[1,2], Roman Jagomägis[1,2], Sven Laur[2].
[1] AS Cybernetica, Institute of Information Security
[2] University of Tartu, Institute of Computer Science

The research reported here was supported by:

1. Estonian Science foundation, grant(s) No. 8058,
2. the target funded theme SF0012708s06 "Theoretical and Practical Security of Heterogenous Information Systems",
3. the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and the Software Technology and Applications Competence Centre, STACC,
4. EU FP6-IST project AEOLUS (contract no. IST-15964).

Cybernetica research reports are available online at
`http://research.cyber.ee/`

Mailing address:
AS Cybernetica
Akadeemia tee 21
12618 Tallinn
Estonia

Privacy-preserving Histogram Computation and Frequent Itemset Mining with Sharemind

Dan Bogdanov, Roman Jagomägis, Sven Laur

January 25, 2010

1 Introduction

1.1 Scope of this document

This report documents computational experiments conducted to estimate the feasibility of privacy-preserving data analysis with the SHAREMIND platform [1]. We chose histogram computation as an example of a standard data mining task. To demonstrate more advanced combinatorial algorithms we also tested two frequent itemset analysis algorithms.

The same results have been reported in a yet unpublished paper [2]. This research report gives the performance results with more technical details and a more thorough analysis.

1.2 Notational conventions

Data types. To explicitly separate private data and public data, we use the following convention. All private variables are in double brackets, e.g. $[\![x]\!]$ denotes the secret shared variable x. The same applies to vectors and matrices, as well. Vectors are denoted by bold lowercase letters ($\boldsymbol{a}, [\![\boldsymbol{b}]\!], \ldots$) and matrices are denoted by uppercase letters ($A, [\![B]\!], \ldots$). Sets and list are denoted by calligraphic uppercase letters ($\mathcal{A}, \ldots$).

Indexing. To simplify notations, we use the semi-standard indexing operator $\mathtt{[}\cdot\mathtt{]}$ to select elements, rows and columns form vectors, lists and matrices. For instance, $\mathcal{A}\mathtt{[5]}$ and $[\![\boldsymbol{b}]\!]\mathtt{[3]}$ denote respectively the 5th element of the list $\mathcal{A}$ and the 3rd element from the vector of shares $[\![\boldsymbol{b}]\!]$. Similarly, $A\texttt{[}2,4\texttt{]}$ and $[\![B]\!]\texttt{[}5,1\texttt{]}$ denote respective elements in the matrices. Additionally, we use index sets and wildcards for brevity and convenience. That is, $A\texttt{[}*,2\texttt{]}$ denotes the second column of the matrix A and $\mathcal{A}\texttt{[}\mathcal{I}\texttt{]}$ denotes a new list $a_{i_1}, \ldots, a_{i_k}$ for an index set $\mathcal{I} = \{i_1, \ldots, i_k\}$. If $\mathcal{I}$ is a zero-one vector of the same length as $\mathcal{A}$ then $\mathcal{A}\texttt{[}\mathcal{I}\texttt{]}$ denotes a new list $a_1, \ldots, a_k$ such that $a_j \in \mathcal{A}\texttt{[}\mathcal{I}\texttt{]}$ iff $\mathcal{I}\texttt{[}j\texttt{]} = 1$ and $a_j \notin \mathcal{A}\texttt{[}\mathcal{I}\texttt{]}$ iff $\mathcal{I}\texttt{[}j\texttt{]} = 0$.

Matrix and vector operations. All operations on matrices are completed elementwise unless specified otherwise. A shorthand $A \odot B$ denotes the elementwise product of matrix entries with confirming dimensions. Let $\boldsymbol{a} \circledast \boldsymbol{b}$ denote the elementwise product of vectors with potentially different lengths, where the shorter vector is repeated to fill out the missing places.

For instance, if $\boldsymbol{a} = (1,2)$ and $\boldsymbol{b} = (1,1,3,5)$, then $\boldsymbol{a} \circledast \boldsymbol{b} = (1,2,3,10)$. The notation can be naturally extended to matrices. Let $\boldsymbol{vec}(A)$ denote a vector that is obtained by stacking all column vectors on top of each other, i.e., $A\texttt{[}*,1\texttt{]}, A\texttt{[}*,2\texttt{]}, \ldots$. Then $A \circledast B = C$ where C has the same dimensions as the larger matrix and $\boldsymbol{vec}(A) \circledast \boldsymbol{vec}(B) = \boldsymbol{vec}(C)$. If matrices have the same row dimensions, then the smaller matrix is replicated as many times to match the size of the larger matrix and elementwise multiplication is performed afterwards.

2 Standard data analysis techniques

2.1 Introduction

Data analysis with Sharemind consists of choosing and implementing the right algorithm for the task. However, choosing the right algorithm with Sharemind is a bit different than with standard architectures.

First, the algorithms of Sharemind should be able to declassify the minimal amount of data. This mostly means that the algorithm should not openly distinguish records in the input data unless this is not a risk to privacy. Typically this means that the programmer must devise an algorithm that processes all the data equally so that no decisions are made from private data. This would allow an attacker to determine these values through flow control analysis.

Second, since parallel processing provides a significant performance boost in Sharemind, algorithms should be chosen based on whether they can be composed from vector operations. Experiments have shown that depending on the computation it may be preferable to choose algorithms that use less operations with more data over algorithms that use more operations with less data.

For example, consider an algorithm that finds the largest value from an array of private values. Typically, one would iterate over all the values and compare each one with a memorized candidate for the largest value. In the end, the candidate value is used as the output. For an n-element array, this requires $O(n)$ operations. For SHAREMIND, we would choose a divide-and-conquer algorithm that recursively divides values into pairs and pushes only the larger values into the next recursion level. In the end, the largest value will remain. This algorithm will require $O(log_2 n)$ comparison operations with different input sizes, as all comparisons on the same level of recursion can be performed in parallel. Due to the smaller number of operations, the recursive algorithm performs significantly better on SHAREMIND when compared to the linear one. This is despite the seemingly larger complexity.

2.2 Histogram computation

A histogram is often the simplest and yet a very powerful way to visualise data and thus used in many data analysis applications. For example, when reporting surveys, graphical histograms are used to describe the distribution of answers to questions with fixed choices. Moreover, if the resolution is properly chosen then the histogram does not disclose much about individual data points.

As a histogram consists of individual bars being proportional to the counts c_i of values in the range $[a_i, b_i]$, the task reduces to finding out how many entries of the vector $\boldsymbol{x}$ are in the range $[a, b]$. If the values are discrete, it may also happen that $a_i = b_i$ and one bar counts the occurrences of a single value.

The tested histogram computation code is straightforward—the private data is passed on directly to the counting code that counts the number of values in a certain range. The counts are declassified and gathered into a vector. The implementation loads the input data from the SHAREMIND private database and runs the histogram function with different value ranges.

Vectorization is used so that a possible value or range is compared to every value in the input data at once. Separate comparison operations are used for testing separate values although even this operation could be vectorized for possibly improved performance. Refer to the `histogram` example application in SHAREMIND 1.91 or later for the SecreC code. The principle of the algorithm is shown in Algorithm 1. The input data is given in the form of a private vector $[\![\boldsymbol{V}]\!]$. The second parameter is the number of possible choices n. Note, that the presented code can easily be adapted to support ranges, if the equality is replaced with two greater-than comparisons.

Algorithm 1 Pseudocode for the SecreC histogram algorithm

```
1:  function HISTOGRAM($[\![\boldsymbol{V}]\!], n$)
2:      for $i \in \{0, \ldots, n-1\}$ do
3:          ⋆⋆⋆ The single value $i$ is duplicated for the elementwise comparison. ⋆⋆⋆
4:          ⋆⋆⋆ If values were equal, the result is one, otherwise it is zero. ⋆⋆⋆
5:          $[\![\boldsymbol{I}]\!] \leftarrow ([\![\boldsymbol{V}]\!] == i)$
6:          $[\![f]\!] \leftarrow$ COLSUM($[\![\boldsymbol{I}]\!]$)
7:          $f \leftarrow$ DECLASSIFY($[\![f]\!]$)
8:          $\boldsymbol{F}_i = f$
9:      end for
10:     return $\boldsymbol{F}$
11: end function
```

Security analysis. To show that an algorithm is private, we have to make sure that the execution flow can be efficiently reconstructed from the public parameters and intended outputs and declassified values can be efficiently computed from the public parameters and intended outputs. While fulfilling these requirement will not directly give us a cryptographic proof, it will show that the private inputs have not been compromised.

For the histogram, it is easy to see that the execution flow of the algorithm depends only on the public input n. All comparisons are performed in parallel and the results are not published. Instead, we make use of the specific comparison operation in SHAREMIND that returns ones and zeroes depending on the result of the comparison. These ones and zeroes are added together to find the number of values that satisfied the equality. Only the number of such values is disclosed, so the algorithm preserves privacy.

2.3 Description of the experiment

We designed the histogram benchmark as follows. We simulated four questions with two, three, five and ten answer choices by creating a table with four columns. Each column contained numeric values from the set $\{0, \cdots, n-1\}$ where n is the number of answer choices. We generated four such tables, containing 100, 1 000, 10 000 and 100 000 values. The application that creates the database and generates data was created using the SHAREMIND controller library.

We then implemented the histogram computation algorithm in SecreC and deployed it in the miners. Another controller application was written that invoked the algorithm, passing input parameters. This application ran the algorithm for each column of each input table.

The miners were set up on a local network connected with a 1 Gb switch. The workstations running the miner software had dual-core Opteron 175 processors and 2 GB of RAM. Linux was used as the operating system. The integrated execution profiling system within the miner software was used to measure the performance of the algorithm.

2.4 Experimental results

The measured timings for histogram computation are given in Table 1.

	2 choices	**3 choices**	**5 choices**	**10 choices**
100 answers	0,76	1,17	1,86	3,99
1 000 answers	2,01	3,59	5,37	10,2
10 000 answers	22,33	42,44	59,39	122,39
100 000 answers	181,53	330,45	524,01	1040,06

Table 1: Timings for histogram computations. Time in seconds.

We notice, that while the timings are linear in the number of choices, they are not strictly linear in the number of inputs. This will become clear once the use of vectorization in the algorithm is explained. The histogram implementation benchmarked performs one vectorized comparison for each possible input value. Given that this comparison takes roughly the same amount of time for each input vector of the same size, it is clear why running the histogram on ten choices is about two times slower than running it on five choices.

However, finding the histogram on 100 000 values is not a thousand times slower from the case with just a hundred values. This is explained by the efficiency of vectorized operations on Sharemind. Since the whole database column is processed at once using a vector operation, Sharemind can optimize its protocol execution to process many values at once.

We conclude the analysis by stating that these benchmarks show that Sharemind could be used for aggregating survey results in acceptable time. While processing a large number of inputs may be time-consuming, most large surveys aggregate no more than a thousand records. And waiting ten seconds for the computation results of a question is perfectly acceptable in non-critical scenarios.

3 Privacy-preserving market basket algorithms

3.1 Frequent itemset mining

Basic definitions. Frequent itemset mining algorithms search co-occurring items, events or actions in transactional data. Let $\mathcal{A} = (a_1, \ldots, a_n)$ be the list of all possible items in the transactional data. Then given m transactions, we can construct a large $m \times n$ zero-one matrix D such that $D[i, j] = 1$ iff a_j is in the ith transaction. Otherwise, $D[i, j] = 0$.

The support of an itemset $\mathcal{X} = \{x_1, \ldots, x_k\}$ is the number of transactions that contain all the items in $\mathcal{X}$. Note that the support can be expressed in terms of multiplications and additions

$$\mathrm{supp}(X) = \sum_{i=1}^{m} \prod_{j \in \mathcal{X}} D[i, j] \tag{1}$$

since the itemset $\mathcal{X}$ is present in the ith row only if $D[i, j] = 1$ for all $j \in \mathcal{X}$. The latter can be expressed with matrix operations

$$\mathrm{supp}(X) = \textsc{Sum}(D[\,*\,, x_1] \odot D[\,*\,, x_2] \cdots \odot D[\,*\,, x_k]) \tag{2}$$

where $\textsc{Sum}(\cdot)$ denotes sum over all matrix elements.

Observe that the vector $D[\,*\,, x_1] \odot D[\,*\,, x_2] \cdots \odot D[\,*\,, x_k]$ indicates which transactions contain $\mathcal{X}$. Therefore, we use a separate shorthand

$$\mathrm{cover}(\mathcal{X}) = D[\,*\,, x_1] \odot D[\,*\,, x_2] \cdots \odot D[\,*\,, x_k] \ . \tag{3}$$

Note that for any two itemsets $\mathcal{X}$ and $\mathcal{Y}$

$$\mathrm{cover}(\mathcal{X} \cup \mathcal{Y}) = \mathrm{cover}(\mathcal{X}) \odot \mathrm{cover}(\mathcal{Y}) \ . \tag{4}$$

The latter means that we can cache some computations and thus can do less multiplications than in the formula (2).

This definition of cover is a bit unconventional, as in the data mining literature the cover usually denotes the set of rows instead of the index vector. However, in the context of privacy-preserving algorithms it is much more efficient to work with index vectors instead of sets.

Properties of frequent itemsets. Which co-occurring itemsets are frequent and which are not is entirely subjective. Given a user specified threshold t, all itemsets $\mathcal{X}$ are considered frequent if $\mathrm{supp}(\mathcal{X}) \geq t$. Clearly, the support is an anti-monotone function $\mathcal{X} \subseteq \mathcal{Y} \Rightarrow \mathrm{supp}(\mathcal{X}) \geq \mathrm{supp}(\mathcal{Y})$ and thus all subsets of a frequent sets must be frequent. This property is known as the Apriori principle. Secondly, note that all frequent itemsets are reachable by iteratively adding single items to discovered frequent itemsets till nothing can be added any more. This observation is known as the pattern growth principle.

3.2 A privacy-preserving version of the Apriori algorithm

Apriori is a frequent itemset mining algorithm, which is based on level-wise search. More precisely, it first generates a list of frequent items $\mathcal{F}_1$ and then runs the following cycle. Given a list of frequent i-element itemsets $\mathcal{F}_i$, it generates a list of potential frequent $(i+1)$-element itemsets $\mathcal{C}_{i+1}$ and verifies which of those are really frequent. Algorithm 2 depicts the pseudo-code of our privacy-preserving version of the Apriori algorithm. The algorithm proceeds similarly except some cover vectors are cached to reduce the number of multiplications. The algorithm also tries to parallelize operations as much as possible. As a result, each iteration of the for loop consists of a single multiplication instruction followed by a single comparison instruction. As the main backside note that the memory consumption of the algorithm increases greatly when the number of frequent itemsets is really large.

The mechanics behind the algorithm is simple though the notation is a bit cryptic. By computing shared column sums over the matrix D, we obtain supports of individual items. By comparing them to the threshold and publishing the corresponding index vector reveals frequent items $\mathcal{F}_1$. Additionally, the covers of $\mathcal{F}_1$ are cached on the line 7. In the body of the loop, we first compute set of plausible $(i+1)$-element frequent items and output two index sets $\mathcal{I}_1$ and $\mathcal{I}_2$ such that

$$\mathcal{C}_{i+1} = \{\mathcal{I}_1\texttt{[}j\texttt{]} \cup \mathcal{I}_2\texttt{[}j\texttt{]} : j = 1, \ldots, |\mathcal{C}_{i+1}|\}$$

Thus, the equation (4) assures that the columns of $M_i\texttt{[}*\texttt{,}\mathcal{I}_1\texttt{]} \odot M_i\texttt{[}*\texttt{,}\mathcal{I}_2\texttt{]}$ correspond to the covers of $\mathcal{C}_{i+1}$. The candidate generation is not different from the standard Apriori algorithm as it is based in the public data. The only new aspect is proper construction of the index sets. Although it is a bit technical, it is still straightforward. Finally, the lines 12–15 count the supports and disclose the frequent itemsets exactly as the lines 5–7.

Algorithm 2 High-level description of privacy-preserving Apriori algorithm

```
1: function Apriori($[\![D]\!], k, t$)
2:     $\star\star\star$ $\mathcal{A}$ is the list of column labels, e.g. $\mathcal{A} = \{1, 2, \ldots, r\}$ $\star\star\star$
3:     $\star\star\star$ $\mathcal{F}$ is the list of currently discovered frequent itemsets $\star\star\star$
4:     $\star\star\star$ $M_i$ is a matrix consisting of cover vectors for the sets $\mathcal{F}_i$ $\star\star\star$
5:     $[\![\boldsymbol{s}]\!] \leftarrow \textsc{ColSum}([\![D]\!])$
6:     $\boldsymbol{f} \leftarrow \textsc{Declassify}([\![\boldsymbol{s}]\!] \geq [\![t]\!])$
7:     $\mathcal{F}_1 \leftarrow \mathcal{A}[\boldsymbol{f}]$, $\mathcal{F} \leftarrow \mathcal{F}_1$, $[\![M_1]\!] \leftarrow [\![D]\!][*, \mathcal{F}_1]$
8:     for $i \in \{1, \ldots, k-1\}$ do
9:         $(\mathcal{C}_{i+1}, \mathcal{I}_1, \mathcal{I}_2) \leftarrow \textsc{GenerateCandidates}(\mathcal{F}_i)$
10:        $[\![M^1_{i+1}]\!] \leftarrow [\![M_i]\!][*, \mathcal{I}_1]$
11:        $[\![M^2_{i+1}]\!] \leftarrow [\![M_i]\!][*, \mathcal{I}_2]$
12:        $[\![M_{i+1}]\!] \leftarrow [\![M^1_{i+1}]\!] \odot [\![M^2_{i+1}]\!]$
13:        $[\![\boldsymbol{s}]\!] \leftarrow \textsc{ColSum}([\![M_{i+1}]\!])$
14:        $\boldsymbol{f} \leftarrow \textsc{Declassify}([\![\boldsymbol{s}]\!] \geq [\![t]\!])$
15:        $\mathcal{F}_{i+1} \leftarrow \mathcal{C}_{i+1}[\boldsymbol{f}]$, $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}_{i+1}$, $[\![M_{i+1}]\!] \leftarrow [\![M_{i+1}]\!][*, \mathcal{F}_{i+1}]$
16:    end for
17:    return $\mathcal{F}$
18: end function
```

Security analysis. Note that the flow of the algorithm depends only on public and declassified variables. Consequently, if we show that all declassified variables can be deduced directly form the desired output, then the protocol is secure—nothing is leaked beyond the output. Indeed, note that given a list of frequent itemsets $\mathcal{F}$ is straightforward to determine the sets $\mathcal{F}_1, \ldots, \mathcal{F}_k$. From these sets it is trivial to compute $\mathcal{C}_2, \ldots, \mathcal{C}_k$ and thus the sizes of matrices nor the index sets $\mathcal{I}_1$ and $\mathcal{I}_2$ leak no additional information than is revealed by the declassification operations. Note that the index vector $\boldsymbol{f}$ can be also computed from the corresponding sets $\mathcal{F}_i$ and $\mathcal{C}_i$. Hence, also the declassification reveals nothing more than $\mathcal{F}_{i+1}$ and the algorithm is secure.

3.3 Privacy-preserving version of the Eclat algorithm

The main drawback of the Apriori algorithm is memory consumption. This issue is amplified even further in the privacy-preserving versions of the algorithm, as we store not only the list of discovered frequent i-element sets but the corresponding cover vectors. As a result, the overall performance is fast but the memory footprint quickly goes beyond reasonable limits. Hence, we also developed a privacy-preserving version of the Eclat algorithm that is based on depth-first search instead of breath-first search. Opposed to the standard implementation, our implementation works with unpacked cover vectors and thus is a bit more inefficient than the original Eclat algorithm.

The Eclat algorithm is based on recursive step that takes in a frequent itemset $\mathcal{X}$ and tries to elongate by adding new items into it. For efficiency reasons, $\textsc{EclatStep}(\cdot)$ takes in a list of potential extensions elements $\mathcal{E}_{\mathcal{X}}$ and the matrix $M_{\mathcal{X}}$ of corresponding cover vectors. Since we want to list only frequent itemsets having less than k elements, the recursion in $\textsc{EclatStep}(\cdot)$ is aborted when the size of $\mathcal{X}$ is larger than k. Otherwise, shares of new cover vectors are computed and the corresponding supports are counted. Finally, $\textsc{EclatStep}(\cdot)$ is applied for each newly frequent itemset and results are merged.

Algorithm 3 High-level description of privacy-preserving Eclat algorithm

```
 1: function $\textsc{EclatStep}(\mathcal{X}, \mathcal{E}_{\mathcal{X}}, [\![M_{\mathcal{X}}]\!], k, t)$
 2:     $\star\star\star$ $\mathcal{F}$ is the list of currently discovered frequent itemsets $\star\star\star$
 3:     $[\![M]\!] \leftarrow [\![M_{\mathcal{X}}]\!][\,*\,,\mathcal{X}\,] \circledast [\![M_{\mathcal{X}}]\!][\,*\,,\mathcal{E}_{\mathcal{X}}\,]$
 4:     $[\![\boldsymbol{s}]\!] \leftarrow \textsc{ColSum}([\![M]\!])$
 5:     $\boldsymbol{f} \leftarrow \textsc{Declassify}([\![\boldsymbol{s}]\!] \geq [\![t]\!])$
 6:     $\mathcal{E} \leftarrow \textsc{Join}(\mathcal{X}, \mathcal{E}_{\mathcal{X}}[\,\boldsymbol{f}\,])$, $\mathcal{F} \leftarrow \mathcal{E}$,
 7:     if $|\mathcal{X}| < k$ then
 8:         for $\mathcal{Y} \in \mathcal{F}$ do
 9:             $\mathcal{E}_{\mathcal{Y}} \leftarrow \textsc{Succ}(\mathcal{E}, \mathcal{Y})$, $M_{\mathcal{Y}} \leftarrow [\![M]\!][\,*\,,\mathcal{E}_{\mathcal{Y}}\,]$
10:             $\mathcal{F} \leftarrow \mathcal{F} \cup \textsc{EclatStep}(\mathcal{Y}, \mathcal{E}_{\mathcal{Y}}, [\![M_{\mathcal{Y}}]\!])$
11:         end for
12:     end if
13:     return $\mathcal{F}$
14: end function

15: function $\textsc{Eclat}([\![D]\!], t, k)$
16:     return $\textsc{EclatStep}(\emptyset, \mathcal{A}, [\![D]\!], k, t)$
17: end function
```

As in the Apriori algorithm the lines 3–6 construct cover vectors, evaluate supports and reveal which of those are over the threshold. Since the algorithm again relies on the equation (4), we have to compute the corresponding sets $\mathcal{X} \cup e$ for all $e \in \mathcal{E}_x[\,\boldsymbol{f}\,]$. The latter is implemented in the function $\textsc{Join}()$. Actually, $\mathcal{E}_{\mathcal{X}}$ is a list of itemsets instead of being a simple list but it works equally well. If $\mathcal{Y}$ is frequent then all one element extensions of $\mathcal{Y}$ can be obtained by joining $\mathcal{Y}$ with other sets in $\mathcal{E}$. Moreover, it is enough if we consider only successor sets of $\mathcal{Y}$ in $\mathcal{E}$. Hence, the call of $\textsc{Succ}(\cdot)$ is used to provide the minimal number of potential extensions for $\mathcal{Y}$.

Security analysis. Again note that the flow of the Eclat algorithm depends only on public parameters and on the declassified $\boldsymbol{f}$ variable. Given the list of frequent itemsets $\mathcal{F}$, it is straightforward to compute the value of $\boldsymbol{f}$. An element in $\boldsymbol{f}\texttt{[}e\texttt{]}$ is one only if the corresponding set $\mathcal{X} \cup e$ is frequent and the latter can be determined form $\mathcal{F}$. Hence, nothing beyond $\mathcal{F}$ is leaked.

3.4 Description of the experiment

In order to test our Apriori and Eclat implementations we used the `mushroom` dataset [3]. The dataset has 8124 transactions with 120 possible items. This dataset was imported into the miners' database by using the `TransactionDataImporter` tool shipped together with SHAREMIND 1.9 and later versions. The tool automatically converts data into the format specified in Section 3.1.

We implemented both algorithms in SecreC and deployed them at the same miner setup detailed in Section 2.3. We used the `ScriptingBenchmark` application to execute the assembly code generated by the SecreC compiler. The algorithm was executed with a variety of desired support parameters ranging from 5000 to 2000.

3.5 Experimental results

In frequent itemset mining the performance is directly linked to the number of candidate itemsets that have to be processed. Since Sharemind cannot leave out transactions that do not contribute to the frequent itemsets due to the privacy requirements, we have to find different paths of optimization. Most importantly, we have to find a way to use parallel execution as much as possible. It is intuitive that a breadth-first algorithm is easier to vectorize than a depth-first algorithm. This is well illustrated by the comparison of Apriori and Eclat runtimes shown in Table 2. The column titles show the support parameter used when starting the algorithm.

With more itemsets, Apriori performs better than Eclat. While this indeed supported our claims about breadth-first algorithms being easier to parallelize, we performed a more thorough analysis by making use of the Sharemind execution profiling mechanism that can measure the structure of algorithm execution and present the results in a table.

	5000	**4000**	**3000**	**2500**	**2000**
Apriori	1,24	1,41	2,13	3,77	17,33
Eclat	1,22	1,74	4,84	10,2	26,46

Table 2: Timings for frequent itemset mining. Time in minutes.

For our experiments we aggregated the results into four categories: private multiplication, private comparison, database operations and other operations. Private multiplications are used in scalar products to join candidate itemsets. Private comparison is used to determine whether an itemset is frequent. Database operations are used to load input data and other operations consist of secret sharing, declassification, private addition and interpreter overhead.

The execution structures for Apriori and Eclat are given in Table 3 and Table 4. We start by noticing that the database timings change little over time. This is because both implementations cache the frequent columns in memory. Also, the database system in the current Sharemind implementation needs optimization, as for higher support values the database turns up to be the most expensive part of the computation.

Support	**Multiplication**	**Comparison**	**Database**	**Other**	**Total**
5000	2,23	1,74	70,15	0,3	74,42
4000	5,87	1,84	76,28	0,53	84,51
3000	27,57	2,42	86,88	11,2	128,06
2500	69,21	3,9	85,56	67,63	226,3
2000	404,13	16,14	88,26	531,05	1039,57

Table 3: Apriori execution structure. Time in seconds.

When we look at the multiplication complexity we see that both algorithms spend similar amounts on multiplication. However, Eclat spends less time in multiplications as it prunes the itemsets differently. We have to consider that private multiplication is significantly less expensive that private comparison, at least in the version of Sharemind used for these benchmarks. Due to this, Eclat loses to Apriori mostly because it is unable to vectorize private comparisons with the same efficiency. And since comparison is expensive, single operations add up to a lot of time.

Finally, we look at the rest of the execution time and see that it grows organically in the case of Eclat but increases sharply for Apriori. This was, interestingly, a side-

Support	**Multiplication**	**Comparison**	**Database**	**Other**	**Total**
5000	1,05	8,45	62,48	1,18	73,15
4000	5,38	27,15	66,74	4,85	104,12
3000	31,49	161,89	68,12	28,93	290,43
2500	75,33	397,22	67,1	72,62	612,27
2000	206,43	1109,15	70,18	201,94	1587,69

Table 4: Eclat execution structure. Time in seconds.

effect of the high level of parallelism. The large vector operations were using up a lot of memory and it caused the servers to run out of physical RAM and start using the swapfile. Optimizations in the memory usage patterns of Sharemind could possibly counter this sharp increase in the execution times and make Apriori scale even better than it currently does.

4 Conclusion

We have presented and in-depth analysis of the SecreC implementations of three algorithms—a histogram algorithm and adaptations of the Apriori and Eclat frequent itemset mining algorithms. We show their SecreC versions preserve privacy and benchmark their performance.

The results show that SecreC can be used to implement well-performing algorithms on the Sharemind privacy-preserving data mining platform. While the histogram implementation is ready for use in real applications, there is room for improvement in the frequent itemset finding performance. Future versions of the Sharemind machine or the algorithms will certainly address the presented issues.

References

[1] Bogdanov, D., Laur, S., Willemson, J., “Sharemind: a framework for fast privacy-preserving computations,” Proc. 13th European Symposium on Research in Computer Security, ESORICS 2008, LNCS 5283, Springer-Verlag, 2008, pp. 192–206.

[2] Bogdanov, D., Jagomägis, R., Laur, S. “Privacy-Preserving Applications in the Hybrid Execution Model”. Unpublished. 2009

[3] Blake, C., Merz, C. "UCI Repository of machine learning databases," University of California, Irvine, Dept. of Information and Computer Sciences, `http://archive.ics.uci.edu/ml/`, 1998.