Project N°: **FP7-284731**

Project Acronym: **UaESMC**

Project Title: **Usable and Efficient Secure Multiparty Computation**

Instrument: **Specific Targeted Research Project**

Scheme: **Information & Communication Technologies**

Future and Emerging Technologies (FET-Open)

Deliverable D4.2.2
Algorithms for large-scale SMC problems

Due date of deliverable: 31st July 2015

Actual submission date: 31st July 2015

Start date of the project: **1st February 2012** Duration: **42 months**

Organisation name of lead contractor for this deliverable: **KTH**

Specific Targeted Research Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Algorithms for large-scale SMC problems

This document summarizes deliverable D4.2.2 of project FP7-284731 (UaESMC), a Specific Targeted Research Project supported by the 7th Framework Programme of the EC within the FET-Open (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at `http://www.usable-security.eu`.

The report presents algorithms developed in workpackage WP4.2 and contains an overview of our research results and an appendix. The overview presents the application scenarios and the existing state of the art and summarizes the algorithms developed in the work package. The appendix consists of several papers and technical reports, each one presenting the technical details of the implemented solutions.

In this deliverable, we report of the following results:

- Implementation and evaluation of differential privacy methods on top of the Sharemind SMC framework. These methods allow us to check that the results of statistical analysis do not contain sensitive information traceable back to particular individuals.
- An algorithm to find the minimum spanning tree of a graph in a privacy-preserving manner
- A string matching algorithm, to find whether a given pattern string is contained in a larger text in a privacy-preserving manner.
- An algorithm to compute shortest distances in sparse graphs, where the lengths of the edges, as well as the structure of the graph itself, are private.
- A privacy preserving algorithm for Frequent Itemset Mining (FIM), which , given a collection of sets, finds the subsets of elements that are present in sufficiently many of these sets.
- Privacy preserving algorithms for Business Process Engineering, which focus on inferring (and checking correctness of) the possible behavior of collaborating (but competitive) enterprises.

List of Authors

Roberto Guanciale (KTH)
Dilian Gurov (KTH)
Peeter Laud (CYB)
Alisa Pankova (CYB)
Martin Pettai (CYB)
Sander Siim (CYB)

Contents

Chapter 1

Introduction

This report gives a review of the algorithms developed during the third year of the UaESMC project. We have investigated several different problems, which required developing of new algorithms, experimenting the different existing algorithms to select the ones suitable for SMC, and driving the development of new SMC techniques described in [3, 5, 25]. Our investigations have been mainly motivated by the example problems selected during the first year [2] and the feedback received by the community and reviewers during the first two years of the project

During the first two years of the project we worked on statistical analysis of structured data [5]. We implemented several analysis preserving "computational privacy": data that must be kept private and there is no single entity that is allowed to see the entire dataset on which the analysis is run. In the last year we investigated another privacy issue: "output privacy". Or goal is to guarantee that the analysis results do not contain sensitive information traceable back to particular individuals. To achieve output privacy, we implemented several differential privacy methods on top of the Sharemind SMC framework. We have implemented a number of statistical functions in this framework, and compared their performance with and without DP mechanisms. Our results show that the extra overhead of implementing DP mechanisms on top of SMC is not prohibitive. Our results are reported in Chapter 2.

In Chapter 3 we describe a method to find the minimum spanning tree of a graph in a privacy-preserving manner. Our proposal makes use of the protocols for concurrent accesses to an array, according to many private indices, that has been described in D2.2.3 [25].

In Chapter 4, we explore algorithms designed for parallel computers in the PRAM model (parallel random-access machine) to build an efficient SMC protocol for privacy-preserving string matching: find whether a private pattern string is contained in a larger and private text.

In Chapter 5 we present an algorithm to compute shortest distances in sparse graphs, where the lengths of the edges, as well as the structure of the graph itself, are private. We make use of the protocol for private lookup that has been described in deliverable D2.2.2 [5, Chapter 4], which, while having linear overhead, pushed almost all complex operations into preprocessing, such that the online communication complexity is constant.

Frequent Itemset Mining (FIM) is a well-known data mining task which consists in, given a collection of sets (transactions), finding the subsets of elements (items) that are present in sufficiently many of these sets. In Chapter 6 we present a privacy preserving algorithm for FIM.

A business process is a collection of structured activities and events that allows an enterprise to achieve a business goal. These processes often involve the structuring of activities of several organizations. This is the case when several potentially competitive enterprises can form a temporary alliance in order to achieve temporary common goals. In Chapter 7 we present privacy preserving techniques for Business Process Engineering that permit the participants to infer the behavior of these collaborations and to check their correctness while keeping secret the participant business processes.

The Appendix is composed of several papers and technical reports that present the technical details of the implemented solutions.

Chapter 2

Combining SMC and Differential Privacy

2.1 Introduction

Many organizations maintain registries that contain private data on the same individuals. Important insights might be gained by these organizations, or by the society, if the data in these registries could be combined and analyzed. The execution of such combination and analysis brings several kinds of privacy problems with it. One of them is *computational privacy* — one must perform computations on data that must be kept private and there is no single entity that is allowed to see the entire dataset on which the analysis is run. Another issue is *output privacy* — it is not *a priori* clear whether the analysis results contain sensitive information traceable back to particular individuals.

Secure Multiparty Computation (SMC) [46, 18] is a possible method for ensuring the computational privacy of a study. To achieve output privacy, the analysis mechanism itself must be designed with privacy in mind [32]. A commonly targeted privacy property is differential privacy (DP) [15, 33], which has both well-understood properties [23] and supports simple arguments, due to its composability.

The main methods of making statistical analysis differentially private are the Laplace and exponential mechanisms, which require the computed function to be sufficiently smooth [15]. For many statistical functions, other mechanisms may provide better accuracy for the same level of privacy. We have chosen the *sample-and-aggregate* method [35], smoothening the function, such that less noise has to be added in order to obtain the same level of privacy.

We report on the experience that we have obtained with the implementations of GUPT's sample-and-aggregate method [33] and the provenance for PDP method [16] on top of the Sharemind SMC framework [8, 9]. We have implemented a number of statistical functions in this framework, and compared their performance with and without DP mechanisms. Our results show that the extra overhead of implementing DP mechanisms on top of SMC is not prohibitive.

2.2 The Sample-and-Aggregate Mechanism

Let us have a dataset T that can be interpreted as the result of $|T|$ times sampling a probability distribution D over **Records** (different samples are independent of each other). By processing T, we want to learn some statistical characteristic $f(D)$ — a vector of values — of the distribution D. We have two conflicting goals — we want to learn this characteristic as precisely as possible, but at the same time we want our processing to be ϵ-differentially private.

A robust method for differentially privately computing the function f is the Sample-and-Aggregate mechanism proposed and investigated by Nissim et al. [35] and Smith [39], and further refined in the GUPT framework [33]. The basic mechanism is given in Alg. 1. Beside the dataset T and the privacy parameter ϵ, Alg. 1 receives as an input a subroutine for computing the function f (without privacy considerations). This subroutine is called by Alg. 1 ℓ times in a black-box manner.

Data: Dataset T, length of the dataset n, number of blocks ℓ, privacy parameter ϵ, clipping range $[\mathsf{left}, \mathsf{right}]$

1 Randomly partition T into ℓ disjoint subsets $T_1, \ldots, T_\ell$ of (almost) equal size
for $i \in \{1, \ldots, \ell\}$ **do**
2 $\quad O_i \leftarrow$ output of the black box on dataset T_i
3 $\quad$ **if** $O_i < \mathsf{left}$ **then** $O_i \leftarrow \mathsf{left}$
4 $\quad$ **if** $O_i > \mathsf{right}$ **then** $O_i \leftarrow \mathsf{right}$
end
5 **Return** $\frac{1}{\ell}\sum_{i=1}^{\ell} O_i + \mathsf{Laplace}\left(\frac{\mathsf{right}-\mathsf{left}}{\ell \cdot \epsilon}\right)$

Algorithm 1: The Sample-and-Aggregate algorithm [33]

We have implemented Alg. 1 on Sharemind. In privacy-preserving statistics applications, before applying an aggregating function, the dataset is usually filtered by some predicate [6, Sec 3]. In the secret-shared setting, we cannot just create a new dataset that contains only the required rows because it would leak the number of rows that matched the predicate. Instead, we must use a mask vector, which contains for each row a boolean that specifies whether this row matched the predicate (and therefore should be used in aggregation) or not. Therefore, all aggregating functions (including the non-differentially private ones used as black boxes in the Sample-and-Aggregate algorithm) receive this mask vector in addition to the dataset and must aggregate only the subset of the dataset denoted by the mask vector.

Also, we needed to replace the **if** statements with oblivious choices and perform the ℓ invocations of the black box in parallel (to get decent performance on Sharemind). Essentially, we needed for each statistical function f that we want to compute, a black box that computes f on any number of datasets (each filtered by a mask vector) in parallel. It is not difficult to implement such black boxes on Sharemind and we have done it for the linear correlation coefficient as an example. For a few other aggregation functions (count, sum, average, median), we have used modified (not fully black-box) algorithms to get better accuracy for the same privacy level.

2.3 Privacy Budgets

When performing statistical analysis, we usually need to make more than one query. We need to guarantee that all performed queries together satisfy a certain privacy property. If several queries are made where the ith query is ϵ_i-differentially private then the composition of the queries is $(\sum \epsilon_i)$-differentially private. We can define a *(global) privacy budget* B and require $\sum \epsilon_i \leq B$. Thus every query consumes a part of the privacy budget and when a query has a higher ϵ than the amount of budget remaining then this query cannot be executed or the accuracy will be reduced.

A global privacy budget has the disadvantage that a query reduces the budget for the whole database even if only a small part of the records participate in the query. To improve this, we can use Personalized Differential Privacy [16]. Here a query can provide a different level of privacy for each record.

We can consider two different methods for enforcing Personalized Differential Privacy. In the simpler case, this means that instead of the global privacy budget, each row in the database has a separate privacy budget. When an ϵ-differentially private query is made, then only the rows participating in the query have their budgets reduced (by ϵ). We call such budgets *in-place budgets* because the budget is stored in the same place (row) as the data to which it applies.

In the more complicated case, each row in the database has a *provenance*, each provenance (not each row) has a privacy budget, and there can be several rows with the same provenance. Thus if an ϵ-differentially private query uses r rows with some provenance p then the budget of this provenance p is reduced by $r\epsilon$. We call such budgets *provenance budgets* because they are stored in a separate table that maps each provenance to its budget. The actual data is accompanied only by the provenance identifier, not the actual budget. We need to use a join operation to connect the values to the budgets.

2.4 Overhead of Differential Privacy

We have implemented queries for all three kinds of privacy budgets described in Sec. 2.3 and compared their performance to the non-differentially private versions of the queries. The implementation uses many primitive operations on secret-shared data. The most important operations (the ones taking most of the running time and communication) for us are integer comparisons, followed by equality checks and multiplications. For smaller data sizes, also floating-point operations are important.

With global budgets, the ratio of the running times of the differentially private query and the non-differentially private one approaches 1 as the data size approaches infinity. Thus the overhead of differential privacy is negligible for large data sizes. For some functions (e.g. linear correlation coefficient), the number of floating-point operations increases from $O(1)$ to $O(\ell)$ (where the parameter ℓ is usually around $\sqrt{n}$, where n is the data size). For small n, this $O(\ell)$ overhead can be significant because floating-point operations on secret-shared data can be several orders of magnitude slower than integer operations.

When comparing the global-budget version of differential privacy with in-place budgets, the extra overhead depends mostly on n, not on the aggregating function. This is because we use the same ϵ-differentially private aggregating functions in both cases but in the latter case we also need to check which rows have enough budget and to reduce the budgets. The overhead here is n comparisons (and n multiplications and n boolean operations, which are much cheaper than comparisons).

Similarly, the extra overhead of the provenance-budgets version of differential privacy compared to the in-place-budgets version depends on data size and not on the aggregating function. Here the data values and the budgets are stored in different tables. Let n_v and n_b be the number of rows in the value table and the budget table, respectively, and $n = n_v + n_b$. Then our algorithm uses $O(n \log n)$ comparisons for sorting (using quicksort) the two tables together and at most a total of $n \log n$ equality checks for another expensive subroutine. The rest of the algorithm is linear-time.

If we need to make several queries in a row on the same value table and the same mask vector (but with possibly different aggregation functions) then we can reuse the results of the $O(n \log n)$ part of the algorithm and need to repeat only the linear-time part for each query. If the next query uses the same value table but a different mask vector then we need to redo the $n \log n$ equality checks. Sorting (the most time-consuming part of our algorithm) needs to be redone only when the next query uses a different value table.

2.5 Conclusion

We have implemented efficient algorithms for performing differentially private statistical analyses on secret-shared data on the SMC platform Sharemind. The current implementation supports the aggregation functions count, sum, arithmetic average, median, and linear correlation coefficent but it can easily be extended to other functions using the Sample-and-Aggregate mechanism. We have implemented three different kinds of budgets for differential privacy and compared their performance. We can conclude that non-trivial queries using various forms of differential privacy can be performed on an SMC platform based on secret sharing, and the performance is good enough to be usable in practice.

Chapter 3

Privacy-preserving Computation of Minimum Spanning Trees

3.1 Oblivious parallel array access

In deliverable D2.2.3 [25] we describe efficient protocols for reading from an array, or writing to an array in parallel, according to many private indices. In this chapter, we show how these protocols can be used to solve a concrete algorithmic task — finding the minimum spanning tree of a graph — in a privacy-preserving manner.

In D2.2.3, we present the following protocols, taking as input, and producing as output the shares of the following values and vectors:

- obliviousRead, taking as input a vector of values $[\![\vec{v}]\!]$ of length m, and a vector of indices $[\![\vec{z}]\!]$ of length n, and returning a shared vector of length n, consisting of the elements of $[\![\vec{v}]\!]$ read from positions given in $[\![\vec{z}]\!]$;

- obliviousWrite, taking as input an existing array $[\![\vec{w}]\!]$ of length m, and vectors of values $[\![\vec{v}]\!]$, of indices $[\![\vec{j}]\!]$, and of priorities $[\![\vec{p}]\!]$, all of length n, and returning an updated array $[\![\vec{w}']\!]$, where the positions given in $[\![\vec{j}]\!]$ are updated with the corresponding values in $[\![\vec{v}]\!]$, and multiple attempts of updating the same position in $[\![\vec{w}]\!]$ are resolved by consulting the priorities in $[\![\vec{p}]\!]$ (only the writing with the highest priority will go through).

Both obliviousRead and obliviousWrite have communication complexity $O((m+n)\log(m+n))$ and round complexity $O(\log(m+n))$, hence they have small overheads (if $n = \Theta(m)$, then the overhead of oblivious array access is only logarithmic in the length of the array). Moreover, both obliviousRead and obliviousWrite are built as the compositions of following protocols:

- $\mathsf{obliviousRead}(m, n, [\![\vec{v}]\!], [\![\vec{z}]\!]) = \mathsf{performRead}(m, n, [\![\vec{v}]\!], \mathsf{prepareRead}(m, n, [\![\vec{z}]\!]))$, where the output of the algorithm prepareRead is an *oblivious shuffle* for vectors of length $m+n$. For the purposes of this chapter, we can consider the output of prepareRead as a value with no further structure.

- $\mathsf{obliviousWrite}(m, n, [\![\vec{w}]\!], [\![\vec{v}]\!], [\![\vec{j}]\!], [\![\vec{p}]\!]) = \mathsf{performWrite}(m, n, [\![\vec{w}]\!], [\![\vec{v}]\!], \mathsf{prepareWrite}(m, n, [\![\vec{j}]\!], [\![\vec{p}]\!]))$, where the output of prepareWrite is a pair of oblivious shuffles for vectors of length $m+n+1$. Again, the structure of the output of prepareWrite does not matter for this chapter.

What matters, are the complexities of these subprotocols. The asymptotic complexities of prepareRead and prepareWrite are the same as for obliviousRead and obliviousWrite — $O((m+n)\log(m+n))$ bits of communication in $O(\log(m+n))$ rounds. But the asymptotic complexities of performRead and performWrite are only $O(m+n)$ bits of communication in $O(1)$ rounds. Hence we try to structure the applications using oblivious parallel array access so, that the amount of calls to prepareRead and prepareWrite is minimized.

3.2 Parallel algorithms for minimum spanning trees

The textbook algorithms for minimum spanning tree (MST) by Kruskal and Prim [11] are inherently sequential, processing the edges or vertices of the graph $G = (V, E, \omega)$, where $\omega : E \to \mathbb{R}$ gives the weights of the edges, one at a time. Hence they are unsuitable for implementation on SMC frameworks based on secret sharing. A lesser-known MST algorithm by Borůvka [34] works as follows:

1. The algorithm is iterative. At the beginning of each iteration, V is partitioned into sets $V_1, \ldots, V_k$ and for each V_i, the MST has already been found.

2. During an iteration, for each V_i we find the edge e_i that has the least weight among the edges that connect a vertex in V_i with a vertex in $V \backslash V_i$.

3. Add all such edges e_i to the MST. Join the parts V_i that are now connected with an edge.

In the beginning, each vertex forms a separate part of V. In the end, all vertices are in a single part. During each iteration, each part is joined with at least one other part, hence the number of parts in the partition will decrease at least by 50%. Hence the number of iterations is at most $\log_2 |V|$. The edges e_i can be found in parallel by considering all edges, and trying to assign some $e \in E$ with ends in V_{j_1} and V_{j_2} to be e_{j_1} and e_{j_2} with priority $-\omega(e)$.

The joining of the parts in step 3 is the hardest to parallelize. A method for this has been proposed by Awerbuch and Shiloach [1], who also slightly adjust the rest of the algorithm. They introduce a union-find data structure, which we can think of as a second set of *directed* edges F (independent of E), where the out-degree of each vertex is 1. Alternatively, we can think of F as a mapping $V \to V$ that maps the source vertex of an F-edge to its target vertex. The graph (V, F) may not contain cycles, except for self-loops, where $F(v) = v$ for some $v \in V$. Hence each connected component of F is a rooted tree that additionally has a self-loop at its root. The connected components of F define the partitioning of V for Borůvka's algorithm.

A connected component of (V, F) is called a *star*, if its height as a tree is at most 1. Hence an isolated vertex is a star, and a component consisting of a root and one or more leaves is also a star. For each vertex v in a star, $F(v)$ is the root of that component. It is easy to check (in parallel), whether a vertex belongs to a star or not: if $F(v) \neq F(F(v))$, then neither v nor $F(F(v))$ are vertices of stars. Also, if afterwards $F(v)$ is not in a star, then v is neither.

In Awerbuch's and Shiloach's adaptation, we consider only such parts V_i during the second step of the algorithm, that are stars according to F. The stars are easy to join with other parts: if v is a vertex in a star that must be joined with the part containing w, then we just make $F(F(v))$ equal to $F(w)$. Such update to F may create cycles of length 2, if two stars were joined with each other, but these are easy to detect and break.

Another important step in Awerbuch's and Shiloach's adaptation is the shortening of F-paths. At the end of each iteration, we update $F(v)$ for all $v \in V$, making it equal to $F(F(v))$. This ensures that a non-star will (eventually) become a star again and will be considered in the second step of Borůvka's algorithm.

These modifications may increase the number of iterations the MST algorithm makes. However, Awerbuch and Shiloach show that it will not grow beyond $\log_{3/2} |V|$.

3.3 Adaption to secret-sharing based SMC

Awerbuch's and Shiloach's PRAM adaption of Borůvka's algorithm can be rather straightforwardly adapted to run on top of a secret-sharing SMC system, e.g. Sharemind. The pseudo-code of our implementation is given in Alg. 3, with the check of a vertex belonging to a star brought out separately in Alg. 2.

While checking for the membership in stars, we have to compare $F(v)$ to $F(F(v))$. Let $G(\cdot) = F(F(\cdot))$. To find G, we have to read from the vector F according to the indices given in F. The actual check takes place in line 3, its results are stored in $[\![\vec{b}]\!]$. In lines 4–6 we assign false to $b_{G(v)}$, where b_v is false. We cannot

Data: Private vector $[\![\vec{F}]\!]$ of length n, where $1 \leq F_i \leq n$
Result: Private predicate $[\![\overrightarrow{St}]\!]$, indicating which elements of $\{1, \ldots, n\}$ belong to stars according to $\vec{F}$

$[\![\sigma]\!] \leftarrow \mathsf{prepareRead}([\![\vec{F}]\!], n)$
$[\![\vec{G}]\!] \leftarrow \mathsf{performRead}([\![\vec{F}]\!], [\![\sigma]\!])$
foreach $i \in \{1, \ldots, n\}$ **do** $[\![b_i]\!] \leftarrow [\![F_i]\!] \stackrel{?}{=} [\![G_i]\!]$;
$[\![b_{n+1}]\!] \leftarrow \mathsf{false}$
foreach $i \in \{1, \ldots, n\}$ **do** $[\![a_i]\!] \leftarrow [\![b_i]\!] \; ? \; (n+1) : [\![G_i]\!]$;
$[\![\vec{b}]\!] := \mathsf{obliviousWrite}([\![\vec{a}]\!], \overrightarrow{\mathsf{false}}, \vec{1}, [\![\vec{b}]\!])$
$[\![\vec{p}]\!] \leftarrow \mathsf{performRead}([\![\vec{b}]\!], [\![\sigma]\!])$ `// Ignore` b_{n+1}
foreach $i \in \{1, \ldots, n\}$ **do** $[\![St_i]\!] \leftarrow [\![b_i]\!] \wedge [\![p_i]\!]$;
return $[\![\overrightarrow{St}]\!]$

Algorithm 2: Privacy-preserving checking for stars

choose, based on b_v, whether we do the assignment or not, but we can choose the target element of the assignment. Hence, if $b_v = \mathsf{true}$, then we assign to $b_{|V|+1}$, which we ignore in the end.

In lines 7–8 we check whether $b_{F(v)}$ is false. If it is, then the end result St_v is also false. We have to read from b according to the indices in F. Hence starCheck contains two private reads according to the indices F. These two reads can share the prepareRead–phase, thereby lowering the complexity of starCheck.

The MST protocol (Alg. 3) starts by initializing $[\![\vec{F}]\!]$, the MST $[\![\vec{\mathcal{T}}]\!]$, and $[\![\vec{\mathcal{W}}]\!]$, where $\mathcal{W}_i$ is used to store i if the i-th edge belongs to the MST. In one iteration, we retrieve the end-points of all edges and check whether they belong to the same part (line 13). If not, and if one of the endpoints of e belongs to a star, then we try to use e as the least-weight edge going out of this star (lines 14–17). The updated $\vec{\mathcal{W}}$ is used to update the result $\vec{\mathcal{T}}$ in line 18. Lines 19–26 combine the breaking of F-cycles of length 2, and the shortening of F-paths. It turns out that in order to find the value of $F(v)$ after these two operations, it is sufficient to know, which of v, $F(v)$, $F(F(v))$ and $F(F(F(v)))$ are equal to each other beforehand. In Alg. 3 we find $G(\cdot) = F(F(\cdot))$ similarly to Alg. 2, and $H(\cdot) = F(F(F(\cdot))) = G(F(\cdot))$. We then perform the necessary comparisons and compute the new value of $F(v)$. The *case*-construction in line 26 is implemented as a composition of oblivious choices.

The asymptotic complexity of our MST protocol is $O(|E| \log^2 |V|)$ communication in $O(\log^2 |V|)$ rounds. More details can be found in [28] (attached to this deliverable).

Data: Number of vertices n, number of edges m
Data: Private vector $[\![\vec{E}]\!]$ of length $2m$ (endpoints of edges, i-th edge is (E_i, E_{i+m}))
Data: Private vector $[\![\vec{\omega}]\!]$ of length m (edge weights)
Result: Private boolean vector $[\![\vec{\mathcal{T}}]\!]$ of length m, indicating which edge belongs to the MST

foreach $i \in \{1, \ldots, 2m\}$ **do**
  $[\![\omega'_i]\!] \leftarrow -[\![\omega_{i \bmod m}]\!]$
  $[\![E'_i]\!] \leftarrow [\![E_{(i+m) \bmod 2m}]\!]$
end
foreach $i \in \{1, \ldots, n+1\}$ **do**
  $[\![F_i]\!] \leftarrow i$
  $[\![\mathcal{W}_i]\!] \leftarrow (m+1)$
end
foreach $i \in \{1, \ldots, m+1\}$ **do** $[\![\mathcal{T}_i]\!] \leftarrow \mathsf{false}$;
$[\![\sigma^{\mathrm{e}}]\!] \leftarrow \mathsf{prepareRead}([\![\vec{E}]\!], n)$
for *iteration_number* $:= 1$ **to** $\lfloor \log_{3/2} n \rfloor$ **do**
  $[\![\overrightarrow{St}]\!] \leftarrow \mathsf{starCheck}([\![\vec{F}]\!])$
  $[\![\vec{F}^{\mathrm{e}}]\!] \leftarrow \mathsf{performRead}([\![\vec{F}]\!], [\![\sigma^{\mathrm{e}}]\!])$ `// Ignore` F_{n+1}
  $[\![\overrightarrow{St}^{\mathrm{e}}]\!] \leftarrow \mathsf{performRead}([\![\overrightarrow{St}]\!], [\![\sigma^{\mathrm{e}}]\!])$
  foreach $i \in \{1, \ldots, m\}$ **do** $[\![d_i]\!] \leftarrow [\![F^{\mathrm{e}}_i]\!] \stackrel{?}{=} [\![F^{\mathrm{e}}_{i+m}]\!]$;
  foreach $i \in \{1, \ldots, 2m\}$ **do** $[\![a_i]\!] \leftarrow [\![St^{\mathrm{e}}_i]\!] \wedge \neg[\![d_{i \bmod m}]\!] \; ? \; [\![F^{\mathrm{e}}_i]\!] : (n+1)$;
  $([\![\sigma^{\mathrm{v}}]\!], [\![\tau^{\mathrm{v}}]\!]) \leftarrow \mathsf{prepareWrite}([\![\vec{a}]\!], [\![\vec{\omega}']\!], n+1)$
  $[\![\vec{F}]\!] := \mathsf{performWrite}([\![\sigma^{\mathrm{v}}]\!], [\![\tau^{\mathrm{v}}]\!], [\![\vec{E}']\!], [\![\vec{F}]\!])$
  $[\![\vec{\mathcal{W}}]\!] := \mathsf{performWrite}([\![\sigma^{\mathrm{v}}]\!], [\![\tau^{\mathrm{v}}]\!], (i \bmod m)_{i=1}^{2m}, [\![\vec{\mathcal{W}}]\!])$
  $[\![\vec{\mathcal{T}}]\!] := \mathsf{obliviousWrite}([\![\vec{\mathcal{W}}]\!], \overrightarrow{\mathsf{true}}, \vec{1}, [\![\vec{\mathcal{T}}]\!])$
  $[\![\sigma^{\mathrm{f}}]\!] \leftarrow \mathsf{prepareRead}([\![\vec{F}]\!], n+1)$
  $[\![\vec{G}]\!] \leftarrow \mathsf{performRead}([\![\vec{F}]\!], [\![\sigma^{\mathrm{f}}]\!])$
  $[\![\vec{H}]\!] \leftarrow \mathsf{performRead}([\![\vec{G}]\!], [\![\sigma^{\mathrm{f}}]\!])$
  foreach $i \in \{1, \ldots, n\}$ **do**
    $[\![c^{(1)}_i]\!] \leftarrow i \stackrel{?}{=} [\![G_i]\!]$
    $[\![c^{(2)}_i]\!] \leftarrow i \stackrel{?}{<} [\![F_i]\!]$
    $[\![c^{(3)}_i]\!] \leftarrow [\![F_i]\!] \stackrel{?}{=} [\![H_i]\!] \wedge [\![F_i]\!] \stackrel{?}{<} [\![G_i]\!]$
    $$[\![F_i]\!] := \begin{cases} i, & \text{if } [\![c^{(1)}_i]\!] \wedge [\![c^{(2)}_i]\!] \\ [\![F_i]\!], & \text{if } [\![c^{(1)}_i]\!] \wedge \neg[\![c^{(2)}_i]\!] \\ [\![F_i]\!], & \text{if } \neg[\![c^{(1)}_i]\!] \wedge [\![c^{(3)}_i]\!] \\ [\![G_i]\!] & \text{if } \neg[\![c^{(1)}_i]\!] \wedge \neg[\![c^{(3)}_i]\!] \end{cases}$$
  end
end
return $([\![\mathcal{T}_1]\!], \ldots, [\![\mathcal{T}_m]\!])$

Algorithm 3: Privacy-preserving minimum spanning tree

Chapter 4

Privacy-preserving String Matching in Parallel

4.1 Introduction

String matching (or string searching) algorithms are an important class of string algorithms that aim to find whether a given pattern string is contained in a larger text. In the public domain, there are many well-known algorithms that solve string matching efficiently, such as the Knuth-Morris-Pratt algorithm [26]. However, our goal is to implement privacy-preserving string matching using secure multi-party computation (SMC) so as to not reveal the contents of the input strings. Possible applications for privacy-preserving string matching include spam e-mail filtering without revealing the e-mail contents to the mail server and finding specific patterns in private genome data.

Many SMC protocols require exchanging information over the network to perform secure computations, and for most state-of-the-art protocols, network communication is the bottleneck for performance. Overall, the communication costs for an SMC protocol can be described with two parameters: *round-complexity* and the *amount of communication* in each round. Round-complexity measures the number of communication rounds the parties have to perform to complete the protocol, where each round of communication has data dependencies with the previous, preventing these rounds to be performed in parallel.

Having a large round-complexity makes a protocol slower due to overhead introduced by network latency. On the other hand, very large network messages are also slow to send since the network bandwidth is bounded. In many situations, a trade-off between round complexity and the amount of communication can be made to gain better overall performance, e.g by increasing round-complexity of the protocol, but reducing the total amount of communication. For example, the authors of [13] show that an SMC protocol with a lower round-complexity, but larger amount of communication is more effective in a high-latency environment, but other methods are optimal when the network latency is smaller.

In this Chapter, we explore algorithms designed for parallel computers in the PRAM model (*parallel random-access machine*) to build an efficient SMC protocol for privacy-preserving string matching. We observe that the PRAM model has analogous criteria for optimizing algorithms as discussed above, minimizing the *time-complexity* of the algorithm (analogous to round-complexity) as well as the *total work* done in all parallel threads (analogous to the amount of communication). Thus, PRAM algorithms are a natural starting point for creating efficient SMC protocols that optimize these two parameters.

Our protocols are built on the Sharemind SMC framework using the `additive3pp` protection domain, which provides a number of arithmetic black-box primitive protocols in a 3-party honest-majority model with semi-honest adversaries [7]. Most operations in `additive3pp` allow for SIMD-like execution, making the transition from PRAM relatively straightforward. As PRAM algorithms require concurrent memory access primitives, we make use of the efficient parallel private array access protocols introduced in [28]. Private array accesses are essential, since otherwise, the algorithms would leak some information about the structure of the input strings. We present a string-matching protocol with optimal trade-off between

round-complexity and amount of communication, which greatly outperforms a brute-force constant-round protocol for already relatively small input sizes in a low-latency environment.

4.2 PRAM Approach to String Matching

We use capital letters A, B to denote strings and use $X[i]$ to refer to the character in string X at position i (starting from 0). Let T be a string of length n and P a string of length m, such that $m \leq n$. The string matching problem is defined as finding all positions in the text T that match with the pattern. That is, we need to calculate the *match* array $\vec{\mathcal{M}}$ of length $n - m + 1$, such that $\mathcal{M}_i = 1$ if and only if P matches T at position i. Our goal is to construct a privacy-preserving string matching protocol that hides both the text and the pattern. Note however that we do not hide the length of the strings.

To implement a privacy-preserving string matching algorithm, one could try to convert for example the Knuth-Morris-Pratt algorithm directly into an SMC protocol. However, the round-complexity of the resulting protocol would be linear in the input sizes $\Theta(m + n)$, which makes it impractical even for small input sizes. Also, there is no natural way to parallelize the algorithm. A second brute-force approach would be to build a protocol which checks for all locations of the input text whether they match the given pattern string. All locations can be tested in parallel with a constant number of rounds by using an equality protocol. However, the number of comparisons that need to be made is $O(m \cdot (n - m))$, which means the total amount of communication required grows quite fast. To build a more efficient protocol, we instead use a PRAM string-matching algorithm described in [24].

The witness array. The used algorithm first pre-processes the pattern string and calculates the *witness* array for the pattern. The witness array is then used to check for matching locations in the text more efficiently. Specifically, given the witness array it is very simple to disqualify candidate positions in the text that definitely do not match the pattern. For a given pattern, the witness array need only be calculated once. We assume the pattern to be non-periodic, that is it cannot be expressed in the form $P = X^k X'$, where X^k is a concatenation of X with itself k times and X' is a prefix of X. The non-periodic case can also be extended to handle periodic patterns.

For a non-periodic pattern P of length m, the witness array $\vec{\mathcal{W}}$ consists of elements $\mathcal{W}_i$ for $i \in \{0, \ldots, \lceil \frac{m}{2} \rceil - 1\}$, such that $P[\mathcal{W}_i] \neq P[\mathcal{W}_i + i]$. That is, the element $\mathcal{W}_i$ indicates a position in P, such that the character at that position differs from the character i positions forward. $\mathcal{W}_0$ is defined to be 0. Since the witness array gives information about the structure of the pattern string, we need to compute it in a privacy-preserving manner. In [38], we give a protocol for calculating the witness array from an input pattern string under secret sharing.

The duel function. The fundamental building-block for the string matching algorithm is the duel function, which, given two indices i, j, disqualifies one as a candidate position in the text to match the pattern. The duel function, presented as Alg. 4, uses the witness array to do this using only a few operations.

Data: Text T of length n, pattern P of length m such that $m \leq n$, witness array $\vec{\mathcal{W}}$ for P and indices i, j such that $j > i$ and $j - i < \lceil \frac{m}{2} \rceil$
Result: One of i or j, eliminating the other as a candidate position for a match
$k \leftarrow \mathcal{W}_{j-i}$
if $T[k + j] = P[k]$ **then**
  return j
else
  return i
end

Algorithm 4: The duel function

Applying the duel function in parallel. The idea of the string-matching algorithm is to divide the input text into blocks of size $\lceil \frac{m}{2} \rceil$, where m is the length of the pattern. Then, a non-periodic pattern can only match the text at most once in each block. In each block, we use the duel function to eliminate all but one possibility of a matching position. Also, we can process each block separately in parallel. Then finally we use the brute-force protocol based on the remaining candidate indices to find the actual match array.

To find the single candidate matching position in each block with the least number of operations, the duel function is applied using a binary tree strategy as depicted on Fig. 4.1.

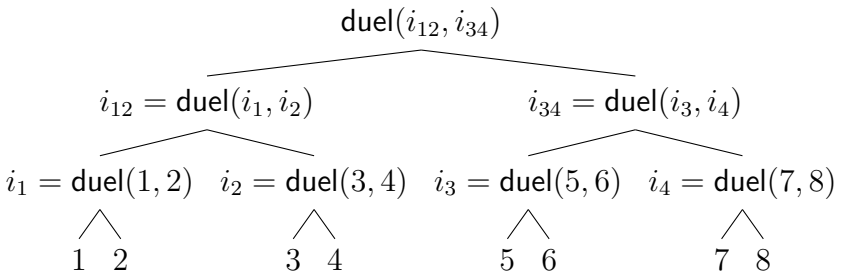


Figure 4.1: Applying binary tree strategy to eliminate all but one position in $\lceil \frac{m}{2} \rceil = 8$ size block

To process each level of the tree with a constant number of rounds, we present a protocol in [38] to apply the duel function in parallel to a set of index pairs. We also present a brute-force string matching protocol which only checks for matchings on a set of private input positions using the private array access primitives from [28].

Applying accelerated cascading. The overall string matching protocol using the binary tree strategy is optimal in the sense of operations made, however the number of rounds is logarithmic in the size of the pattern. A faster protocol is obtained by using a *doubly-logarithmic* tree instead, which has height $\log \log k + 1$ for k leaves, but uses more operations. To still process each level of the doubly-logarithmic tree in constant rounds, we additionally need a protocol applying the duel function to a set of indices in parallel and returning a single potential matching candidate. This can be done by simply to applying duel to all pairs of indices in the input set.

Finally, an optimal asymptotic trade-off between round-complexity and communication is achieved by using a combination of both. First, the binary tree is used up to $\lceil \log \log \log \frac{m}{2} \rceil$ levels and then switching to the doubly-logarithmic tree. The corresponding PRAM algorithm has time complexity $O(\log \log m)$ and work complexity $O(n)$. This method of combining different parallel execution strategies is called *accelerated cascading*, and as a general method, can be applied to any operation for which an algorithm exists that can process each level of the execution tree in constant time.

Our experiments show, that the brute-force constant-round protocol is faster for smaller input sizes, which is most likely due to the overhead of the private array access primitives used in the PRAM protocol. However, for larger input sizes, the PRAM approach achieves much better performance.

4.3 Conclusion

The string matching protocol presented in [38] shows that the PRAM model offers useful insight into designing more efficient SMC protocols. The accelerated cascading design principle allows to effectively fine-tune the trade-off between round-complexity and total communication of the protocol for different network environments. Using PRAM methods, we build an efficient SMC protocol for string matching without revealing any information about the input strings. The performance of the protocol relies on the existence of efficient private array access primitives from [28]. Similarly, many other classes of problems for which an efficient PRAM algorithm exists can be solved in a privacy-preserving manner by building an equivalent SMC protocol.

Chapter 5

Privacy-preserving Finding of Shortest Distances in Sparse Graphs

In deliverable D2.2.2 [5, Chapter 4], we reported on a protocol for private lookup, which, while having linear overhead, pushed almost all complex operations into preprocessing, such that the online communication complexity is constant [27] (the paper is attached to this deliverable). We have applied this technique to compute shortest distances in sparse graphs, where the lengths of the edges, as well as the structure of the graph itself, are private.

5.1 The private lookup protocol

Let us recall the construction of the protocol, we refer to [27] for more details. Let $(\llbracket v_1 \rrbracket, \ldots, \llbracket v_n \rrbracket)$ be a vector of private values, where $v_1, \ldots, v_n$ are elements of some finite field $\mathbb{F}$. We also interpret the indices $1, \ldots, n$ as *non-zero* elements of $\mathbb{F}$. The representation $\llbracket \cdot \rrbracket$ of private values must be based on some kind of secret sharing over the field $\mathbb{F}$, such that the addition of two private values, and the multiplication of a private value with a public value are free operations, i.e. they require no communication between different computing parties. Let $\llbracket j \rrbracket$, where $j \in \{1, \ldots, n\}$, be the private index; we want to find $\llbracket v_j \rrbracket$.

There exists a polynomial f over $\mathbb{F}$ of degree at most $n-1$, such that $f(i) = v_i$ for all $i \in \{1, \ldots, n\}$. Using Lagrange interpolation, the coefficients $c_0, \ldots, c_{n-1}$ of this polynomial can be computed as certain linear combinations of $v_1, \ldots, v_n$. The multipliers in these linear combinations only depend on n and $\mathbb{F}$. Hence the computing parties can compute $\llbracket c_0 \rrbracket, \ldots, \llbracket c_{n-1} \rrbracket$ using free operations only.

To find $\llbracket v_j \rrbracket$, the computing parties first somehow have to obtain $\llbracket j \rrbracket^2, \ldots, \llbracket j \rrbracket^{n-1}$ from $\llbracket j \rrbracket$. Afterwards, they can compute $\llbracket v_j \rrbracket = \sum_{i=0}^{n-1} \llbracket c_i \rrbracket \llbracket j \rrbracket^i$. If the representation $\llbracket \cdot \rrbracket$ is based on Shamir's secret sharing [37], then the computation of this scalar product only costs as much (in communication) as the computation of single multiplication of two private values [17]. In particular, the communication cost of computing the scalar product does not depend on n.

To obtain $\llbracket j \rrbracket^2, \ldots, \llbracket j \rrbracket^{n-1}$, the computing parties proceed in a roundabout manner. They generate $\llbracket r \rrbracket$, where r is a uniformly randomly chosen non-zero element of $\mathbb{F}$. Together with $\llbracket r \rrbracket$, they also generate $\llbracket r \rrbracket^{-1}$, as described in [12]. They compute $\llbracket r \rrbracket^2, \ldots, \llbracket r \rrbracket^{n-1}$ in a straightforward manner, using $O(n)$ multiplications. To compute the powers of $\llbracket j \rrbracket$, they first compute a public value $z \in \mathbb{F} \backslash \{0\}$ as a declassification of $\llbracket j \rrbracket \cdot \llbracket r \rrbracket^{-1}$. It is safe to declassify this value, because r^{-1} is a uniformly randomly distributed element of $\mathbb{F} \backslash \{0\}$ and thus perfectly hides j. Afterwards, the parties can compute $\llbracket j \rrbracket^i = z^i \cdot \llbracket r \rrbracket^i$ for $i \in \{2, \ldots, n-1\}$, using only free operations.

To generate $\llbracket r \rrbracket$ and $\llbracket r \rrbracket^{-1}$, and to compute the powers of $\llbracket r \rrbracket$, no knowledge of $\llbracket v_1 \rrbracket, \ldots, \llbracket v_n \rrbracket$ nor $\llbracket j \rrbracket$ is necessary. Hence, these computations can be performed off-line, leaving only the computation of $\llbracket j \rrbracket \cdot \llbracket r \rrbracket^{-1}$, its declassification, and the computation of the scalar product $\sum_{i=0}^{n-1} \llbracket c_i \rrbracket \llbracket j \rrbracket^i$ as non-free online computations. The online computations require a constant amount of communication; it does not depend on n.

5.2 Protocol for SSSD

Let $G = (V, E)$ be a directed graph with $s, t : E \to V$ giving the source and target, and $w : E \to \mathbb{N}$ giving the length of each edge. Let $v_0 \in V$ be the vertex, the distance of which from other vertices we're interested in. Bellman-Ford (BF) algorithm for SSSD starts by defining $d_0[v] = 0$, if $v = v_0$, and $d_0[v] = \infty$ for $v \in V \backslash \{v_0\}$. It will then compute $d_{i+1}[v] = \min(d_i[v], \min_{e \in t^{-1}(v)} d_i[s(e)] + w(e))$ for all $v \in V$ and $i \in \{0, \ldots, |V| - 2\}$. The vector $\vec{d}_{|V|-1}$ is the result of the algorithm.

We have implemented the BF algorithm on top of the Sharemind platform, hiding the structure of the graph, as well as the lengths of edges. In our implementation, the numbers $n = |V|$ and $m = |E|$ are public, and so are the in-degrees of vertices (obviously, these could be hidden by using suitable paddings). In effect, the mapping t in the definition of the graph is public, while the mappings s and w are private. During the execution, we use private lookup to find $d_i[s(e)]$. This is the only array access operation, where the array index is private. An advantage of the BF algorithm over Dijkstra's algorithm is, that the former has no array writing operations with private indices.

We have attempted to parallelize our privacy-preserving implementation of the BF algorithm. In general, the SSSD algorithms do not parallelize well. In our case, the vectors $[\![\vec{d}_1]\!], \ldots, [\![\vec{d}_{|V|-1}]\!]$ have to be computed one after another. On the other hand, all elements of a given vector $[\![\vec{d}_i]\!]$ can be computed in parallel in a constant number of rounds. Hence our implementation has $O(n)$ rounds in the online phase.

Chapter 6

Privacy-preserving Frequent Itemset Mining Using Sparse Set Representations

Frequent Itemset Mining (FIM) is a well-known data mining task whose privacy-preserving version has also been considered [4, 10, 30, 48, 40]. The task is, given a collection of sets (*transactions*), find the subsets of elements (*items*) that are present in sufficiently many of these sets. After finding which items are more likely to occur together, one may search for the reason for that co-occurrence, and whether the existence of one item implies the existence of the other one (associative rule mining).

Traditionally, the sets themselves are called the transactions, and their elements are called items. This comes from one possible use case, where the items are some goods sold in the supermarket, and each transaction corresponds to the contents of one shopping cart that a client has bought.

6.1 Data Representation

In the standard setting, the input database is represented by a bit matrix, where the rows are characteristic vectors of transactions over the universal set of items, where each column corresponds to a certain item. In practice, this matrix is often very sparse and contains significantly more zeroes than ones. The previous privacy-preserving FIM protocols have not tried to benefit from this sparsity. In a perfectly secure setting, this is simply impossible, since a dense dataset has to be indistinguishable from a sparse dataset. If we consider density as not very sensitive information, we may improve the efficiency by developing specific algorithms that take sparsity into account. Developing such algorithms is the main contribution of this work. Its full version can be found in [29], which is also included in Appendix E.

6.2 Notation

Throughout this paper, we use the following quantities:

- m is the number of rows in the data table (transactions);
- n is the number of columns in the data table (items); the same notation is used to denote the number of columns on current iteration (itemsets);
- $m_j \leq m$ is the number of non-zero entries in the j-th column;
- $m' \geq m_j (\forall j)$ is the upper bound on the number of non-zero entries in each column;
- t is the threshold of being a frequent set;
- k is the size of the currently generated itemsets;
- $\hat{k}$ is the maximal size of frequent itemsets that have to be generated.

Some FIM-specific notation:

- $\sigma(I)$ is the set of all transactions containing the itemset I (the support of I);
- $\Delta(I_1, I_2) := \sigma(I_1) \setminus \sigma(I_2)$ is the difference of the supports of I_1 and I_2.

The following more general shorthand operation notation will be used:

- protocol input length (if the input is a vector) n;
- number of bits of protocol input: k;
- secret shared value (additive or xor sharing) $\langle\!\langle a \rangle\!\rangle$;
- additively shared value $[\![a]\!]$;
- xor shared value $\langle\langle a \rangle\rangle$;
- i-th element of a vector $\mathbf{a}$: a_i and $\mathbf{a}[i]$;
- (i, j)-th element of a matrix $\mathbf{A}$: (a_{ij}) and $\mathbf{A}[i, j]$;
- vector elements from i-th to j-th: $\mathbf{x}[i : j]$
- vector concatenation: $\mathbf{x} \| \mathbf{y}$;
- $[1]_m = \overbrace{[1, \ldots, 1]}^{m}$;
- zip of two (equal-length) vectors: $\mathbf{x} \bowtie \mathbf{y} = [(x_i, y_i) \mid i \leftarrow [1, \ldots, |\mathbf{x}|]]$.
- matrix columnwise multiplication: $\mathbf{A} \otimes \mathbf{B}$. Namely, if $\mathbf{A} = (\mathbf{a}_1 \| \ldots \| \mathbf{a}_{n_A})$ and $\mathbf{B} = (\mathbf{b}_1 \| \ldots \| \mathbf{b}_{n_B})$, then $\mathbf{A} \otimes \mathbf{B} = (\mathbf{c}_{1,1}, \ldots, \mathbf{c}_{n_A, n_B})$ where $\mathbf{c}_{i,j}[k] = \mathbf{a}_i[k] \cdot \mathbf{b}_j[k]$;
- protocol composition:
  - $P1 + P2$ execute P_1 and P_2 sequentially;
  - $n \odot P$ execute n instances of P in parallel;

6.3 Main Algorithms

There exist several variations for the standard FIM algorithms. In this section, we just give the intuition about how these algorithms work, without introducing particular algorithm descriptions, as the actual implementations may vary.

Apriori This algorithm sequentially constructs all the frequent itemsets of size 1, then of size 2, until all the frequent sets of size $\hat{k}$. Any infrequent itemsets are immediately discarded. The frequent sets of size k are constructed only for those sets whose all $k - 1$ subsets have been frequent. The straightforward implementation of this algorithm does not keep in memory the lists of transactions that contain sets of size other than 1, and on each iteration, the sets are constructed from the initial database. The way in which these sets are constructed depends on the particular algorithm instance. One possible recursive implementation of Apriori is given in Alg. 5.

```
Data: M all the frequent sets of size k − 1
Result: Frequent itemsets of size at least k
F ← ∅ ;
foreach X_i ∈ M do
    foreach X_j ∈ M, j > i do
        R ← X_i ∪ X_j ;
        if |R| ≥ t then
            F ← F ∪ {R} ;
        end
    end
end
if F ≠ ∅ then
    F′ ← Apriori(F) ;
end
return F ∪ F′ ;
```

Algorithm 5: Apriori

```
Data: [P] all the frequent sets of size k − 1 with a prefix P
Result: Frequent itemsets of size at least k with the prefix P
foreach X_i ∈ [P] do
    F_i ← ∅ ;
    foreach X_j ∈ [P], j > i do
        R = X_i ∪ X_j ;
        if |R| ≥ t then
            F_i = F_i ∪ {R} ;
        end
    end
    if F_i ≠ ∅ then
        F′_i = Eclat(F_i) ;
    end
end
return ⋃_i F′_i ;
```

Algorithm 6: Eclat

```
Data: [P] all the frequent sets of size k − 1 with a prefix P
Result: Frequent itemsets of size at least k with the prefix P
foreach X_i ∈ [P] do
    F_i ← ∅ ;
    foreach X_j ∈ [P], j > i do
        X_ij ← X_i ∪ X_j ;
        Δ(X_i, X_ij) ← Δ(P, X_j) \ Δ(P, X_i) ;
        |σ(X_ij)| = |σ(X_i)| − |Δ(X_i, X_ij)| ;
        if |σ(X_ij)| ≥ t then
            F_i = F_i ∪ {X_ij} ;
        end
    end
    if F_i ≠ ∅ then
        F'_i = Diffset(F_i) ;
    end
end
return ⋃_i F'_i ;
```

Algorithm 7: Diffset

Eclat Similarly to Apriori, this algorithm constructs sets of size k from sets of size $k-1$. The main difference from Apriori is that this algorithm uses depth-first search, considering on one iteration not all the possible subsets of size k, but rather constrains it to the sets with a common *prefix* P of length $k-1$ (these are all sets of the form $P \cup \{x\}$ for $x \notin P$). Let the support of P be denoted $\sigma(P)$. For each item x, all possible frequent sets with prefix $P' := P \cup \{x\}$ can be constructed as $\sigma(P \cup \{x\}) \cap \sigma(P \cup \{y\})$ for all other sets $(P \cup \{y\})$, $y \neq x$. The prefix P' is then processed recursively. The description is given in Alg. 6.

Diffset If the matrix columns are dense, then instead of keeping a set of transactions that *contain* the given itemset, one could try to keep a set of transactions that *do not contain* the given itemset. Another FIM algorithm Diffset [47] is similar to Eclat, but instead of keeping the set of transactions in each itemset, it keeps just the *sizes of supports* of sets of size $k-1$, and the differences between a set of size k and its subsets of size $k-1$. In this way, even if the initial matrix is not dense, the algorithm may still give better efficiency on later iterations.

Let the itemsets $P \cup \{x\}$ and $P \cup \{y\}$, be frequent. The question is whether the itemset $P \cup \{x\} \cup \{y\}$ is frequent. Let $\Delta(P \cup \{x\}, P \cup \{x\} \cup \{y\})$ be the difference between the supports $\sigma(P \cup \{x\})$ and $\sigma(P \cup \{x\} \cup \{y\})$. We have $\sigma(P \cup \{x\} \cup \{y\}) = \sigma(P \cup \{x\}) \setminus \Delta(P \cup \{x\}, P \cup \{x\} \cup \{y\})$, so the size of the support is $|\sigma(P \cup \{x\} \cup \{y\})| = |\sigma(P \cup \{x\})| - |\Delta(P \cup \{x\}, P \cup \{x\} \cup \{y\})|$, and $\Delta(P \cup \{x\}, P \cup \{x\} \cup \{y\})$ can be computed as $\Delta(P, P \cup \{y\}) \setminus \Delta(P, P \cup \{x\})$. The description is given in Alg. 7.

The main step of presented privacy-preserving FIM algorithms We now make a small summary of the privacy-preserving FIM algorithms proposed above. A similar property of these algorithms is that, on each step of each iteration, all they compute a frequent itemset of size k, based on the frequent itemsets of size $k-1$. The basis of finding a k-set from $k-1$ subsets is either *set intersection* (for Apriori and Eclat), or *set difference* (for Diffset).

A straightforward approach to find a set intersection is to represent each set as a characteristic bit vector of the item universe, and then find the pointwise product of these vectors. This method is used in the bit matrix representation.

Although all matrix elements are bits, at some moment the column sum has to be computed. In this way, in [4] the matrix elements are initially all at least $\log m$-bit, since the maximal value that the sum may take is m. For very sparse sets, such an encoding may be excessive due to large amount of zeroes that will

Algorithm Call	Description	Returned Value
$\mathsf{Csort}(\llbracket\mathbf{x}\rrbracket \bowtie \llbracket\mathbf{b}\rrbracket)$	counting sort	$\llbracket\mathbf{y}\rrbracket$ where $\mathbf{y}$ is $\mathbf{x}$ sorted by bit keys $\mathbf{b}$
$\mathsf{Rsort}(\llbracket\mathbf{x}\rrbracket)$	radix sort	$\llbracket\mathbf{y}\rrbracket$ where $\mathbf{y}$ is sorted $\mathbf{x}$
$\mathsf{Qsort}(\llbracket\mathbf{x}\rrbracket)$	quick sort	$\llbracket\mathbf{y}\rrbracket$ where $\mathbf{y}$ is sorted $\mathbf{x}$
$\mathsf{CountOnes}(\langle\langle\mathbf{x}\rangle\rangle)$	count ones in a bit vector	$\langle\langle c\rangle\rangle$ s.t $c = \sum_{i=1}^{n} x_i$, $x_i \in \mathbb{Z}_2$
$\mathsf{CharVec}(\langle\langle a\rangle\rangle, k)$	characteristic vector	2^k-bit vector $\langle\langle\mathbf{b}\rangle\rangle$ s.t $b_i = 1$ iff $i = a$
$\mathsf{MultByBit}(\langle\langle a\rangle\rangle, \langle\langle b\rangle\rangle)$	multiply by bit	$\langle\langle 0\rangle\rangle$ if $b == 0$ and $\langle\langle a\rangle\rangle$ otherwise

Table 6.1: Building block operations

not be needed anyway. In [4], finding an intersection of two itemsets i and j and checking its cardinality is implemented as:

1. multiply pointwise two $\log m$-bit vectors of length m;
2. sum the obtained m products up;
3. compare the obtained $\log m$-bit number with a $\log m$-bit number t.

Another possibility to do the same thing is to keep all the bits in $\mathbb{Z}_2$, doing the share conversion *after* the multiplication. This approach can be useful if the share conversion protocol is implemented more efficiently than multiplication.

6.4 Set Based Approach

Let each column of the matrix contain at most m' entries for $m' \leq m$. We will now use an $m' \times n$ matrix for data table representation. Each column will now be not the characteristic bit vector, but will contain the indices of transactions straightforwardly. Encoding a number from $[1, \ldots, m]$ requires $\log m$ bits. If the table contains at most nm' nonzero entries, then $nm' \cdot \log m$ bits are sufficient to encode it. If the size of some column is $m_j < m'$, then some $m' - m_j$ of its entries are set to 0. The order of values in a column does not matter.

If the columns are sparse, we get $m' \ll m$, and hence the set representation allows to save a lot of memory space. We also need to develop efficient algorithms for set intersection and difference.

6.4.1 Building Block Algorithms

Our set operation algorithms in turn use some smaller more general subalgorithms, some of which have already been implemented for Sharemind before, and some had to be implemented as a part of this work. The description of these algorithms is given in Tab. 6.1, and their complexity (expressed in terms of existing Sharemind protocols) in Tab. 6.2. The details of implementations of some of these algorithms (that are part of this work) and references to implementations of some other algorithms (that are not a part of this work) can be found in [29].

6.4.2 Set intersection of k-bit elements

Let $\mathbf{a}$ and $\mathbf{b}$ be the two arrays that represent the sets whose intersection we are going to find. Let $|\mathbf{a}| = n_1$, $|\mathbf{b}| = n_2$, $n = n_1 + n_2$. The computation of $\mathbf{c} = \mathbf{a} \cap \mathbf{b}$ is given in Alg. 8.

First, the algorithm sorts the straightforward concatenation $\llbracket\mathbf{a}\rrbracket \| \llbracket\mathbf{b}\rrbracket$ by value, so that if an element occurs in both sets, then these two elements appear together in the resulting sorted array. Hence if there are two sequential instances of the same element d_{i-1} and d_i in $\mathbf{d}$, then we chose $s_i = d_i$. Everywhere else $s_i = 0$. In this way, we keep exactly those elements that are present in both $\mathbf{a}$ and $\mathbf{b}$. We sort the elements

Algorithm	Secure Operation Complexity
$\overline{\mathsf{Csort}}(n,k)$	$n \odot \overline{\mathsf{Mult}}(k) + n \odot \overline{\mathsf{ShareConv}}(k) + \overline{\mathsf{Shuffle}}(n,2k) + n \odot \overline{\mathsf{Declassify}}(k)$
$\overline{\mathsf{Rsort}}(n,k)$	$k \cdot (n \odot \overline{\mathsf{Mult}}(k) + n \odot \overline{\mathsf{ShareConv}}(k) + \overline{\mathsf{Shuffle}}(n,2k) + n \odot \overline{\mathsf{Declassify}}(k))$
$\overline{\mathsf{Qsort}}(n,k)$	$\overline{\mathsf{Shuffle}}(n,k) + \log n \cdot (n \odot \overline{\mathsf{LessThan}}(k) + n \odot \overline{\mathsf{Declassify}}(k))$
$\overline{\mathsf{CountOnes}}(n,k)$	0
$\overline{\mathsf{CharVec}}(k)$	$k \cdot \overline{\mathsf{OuterProd}}(\sqrt{2^k},1)$
$\overline{\mathsf{MultByBit}}(k)$	$\overline{\mathsf{OuterProd}}(k+1,1)$.

Table 6.2: Complexity of building block operations

Data: $\llbracket \mathbf{a} \rrbracket$ and $\llbracket \mathbf{b} \rrbracket$ where all elements except 0 are unique
Result: $\llbracket \mathbf{c} \rrbracket = \llbracket \mathbf{a} \cap \mathbf{b} \rrbracket$, $|\mathbf{c}| = \min(|\mathbf{a}|, |\mathbf{b}|)$

$\llbracket \mathbf{d} \rrbracket \leftarrow \mathsf{Sort}(\llbracket \mathbf{a} \rrbracket \| \llbracket \mathbf{b} \rrbracket)$;
$\llbracket t_1 \rrbracket \leftarrow 0$;
$\llbracket s_1 \rrbracket \leftarrow 0$;
foreach $i \in \{1, \ldots, |\mathbf{a}| + |\mathbf{b}| - 1\}$ **do**
  $\llbracket t_i \rrbracket = \mathsf{Equal}(\llbracket d_i \rrbracket, \llbracket d_{i-1} \rrbracket)$;
  $\llbracket s_i \rrbracket = \mathsf{MultByBit}(\llbracket t_i \rrbracket, \llbracket d_i \rrbracket)$;
end
$\llbracket \mathbf{c} \rrbracket = \mathsf{Csort}(\llbracket \mathbf{t} \rrbracket \bowtie \llbracket \mathbf{s} \rrbracket)$;
return $\llbracket \mathbf{c} \rrbracket[0 : \min(|\mathbf{a}|, |\mathbf{b}|), 1]$;

Algorithm 8: Set intersection $\mathsf{Set}_\cap$

once more according to the bits $\mathbf{t}$ in order to get all the zeroes into the end of the array, so that the excessive zeroes could be safely removed. The intersection contains at most $\min(n_1, n_2)$ elements.

We have $|\mathbf{s}| = |\mathbf{c}| = |\mathbf{a}| + |\mathbf{b}| = n$. The iterations of the for-cycle do not depend on each other and hence are parallelizable. The number of used operations is $\overline{\mathsf{Sort}}(n,k) + n \odot (\overline{\mathsf{Equal}}(k) + \overline{\mathsf{MultByBit}}(k)) + \overline{\mathsf{Csort}}(n,k)$. Sort can be instantiated either to Rsort or Qsort, and one may be preferable to the other depending on the parameters and whether we want to win more in rounds or in communication.

6.4.3 Set difference of k-bit elements:

The algorithm is analogous to set intersection. The computation of $\mathbf{c} = \mathbf{a} \setminus \mathbf{b}$ is given in Alg. 9. The difference is that now we should leave exactly the elements that are in $\mathbf{a}$, but not in $\mathbf{b}$. In order to do this, we add a bit 1 to each element of $\mathbf{a}$ and a bit 0 to each element of $\mathbf{b}$, so that now we sort pairs of elements. After sorting, if two elements are equal, then the bit 0 comes before 1. Now if two sequential elements are the same in $\mathbf{c}$, we set $t_i = 0$. Otherwise, we set $t_i = 1$ unless the element comes from the second set (has the label 0).

Adding a bit to $\mathbf{a}$, $\mathbf{b}$ and removing it from $\mathbf{d}$ is free in any secret sharing scheme since we treat this concatenation just as a pairing. The number of used operations is almost the same as for the set intersection, except one extra bit in $\overline{\mathsf{Sort}}(n, k+1)$ that adds a negligible complexity overhead compared to $\overline{\mathsf{Set}}_\cap$.

The summary of set operation complexities is given in Tab. 6.3.

6.4.4 Mixed Columns

If some columns of a sparse matrix are dense, then our set based algorithm can be much slower than ordinary bit vector intersection. If there is just a single item that is contained in all m transactions, then $m' = m$, and we do not reduce the table size at all. Even if we agree to leak precise density of each column, it may happen that there are only a few dense columns in the beginning, but their number increases after a couple of iterations, as the sparsest columns will be just filtered out due to not reaching the threshold. We might

Data: $\llbracket \mathbf{a} \rrbracket$ and $\llbracket \mathbf{b} \rrbracket$ where all elements except 0 are unique
Result: $\llbracket \mathbf{c} \rrbracket = \llbracket \mathbf{a} \setminus \mathbf{b} \rrbracket$, $|\mathbf{c}| = |\mathbf{a}|$

$\llbracket \mathbf{a}' \rrbracket := \llbracket \mathbf{a} \rrbracket \bowtie [1]_{|\mathbf{a}|}$;
$\llbracket \mathbf{b}' \rrbracket := \llbracket \mathbf{b} \rrbracket \bowtie [0]_{|\mathbf{b}|}$;
$\llbracket \mathbf{d} \rrbracket \leftarrow \mathsf{Sort}(\llbracket \mathbf{a}' \rrbracket \| \llbracket \mathbf{b}' \rrbracket)$;
$\llbracket t_1 \rrbracket \leftarrow 0$;
$\llbracket s_1 \rrbracket \leftarrow 0$;
foreach $i \in \{1, \ldots, |\mathbf{a}| + |\mathbf{b}| - 1\}$ **do**
  $\llbracket t_i \rrbracket = \mathsf{Equal}(\llbracket d_i \rrbracket[0], \llbracket d_{i-1} \rrbracket[0]) - \llbracket d_i \rrbracket[1]$;
  $\llbracket s_i \rrbracket = \mathsf{MultByBit}(\llbracket t_i \rrbracket, \llbracket d_i \rrbracket[0])$;
end
$\llbracket \mathbf{c} \rrbracket = \mathsf{Csort}(\llbracket \mathbf{t} \rrbracket \bowtie \llbracket \mathbf{s} \rrbracket)$;
return $\llbracket \mathbf{c} \rrbracket[0 : |\mathbf{a}|, 1]$;

Algorithm 9: Set difference $\mathsf{Set}_\setminus$

Algorithm Call	Returned Value	Secure Operation Complexity
$\mathsf{Set}_\cap(\llbracket \mathbf{a} \rrbracket \| \llbracket \mathbf{b} \rrbracket)$	$\llbracket \mathbf{c} \rrbracket = \llbracket \mathbf{a} \cup \mathbf{b} \rrbracket$	$\overline{\mathsf{Sort}}(n,k) + n \odot (\overline{\mathsf{Equal}}(k) + \overline{\mathsf{MultByBit}}(k)) + \overline{\mathsf{Csort}}(n,k)$
$\mathsf{Set}_\setminus(\llbracket \mathbf{a} \rrbracket \| \llbracket \mathbf{b} \rrbracket)$	$\llbracket \mathbf{c} \rrbracket = \llbracket \mathbf{a} \setminus \mathbf{b} \rrbracket$	$\overline{\mathsf{Sort}}(n,k+1) + n \odot (\overline{\mathsf{Equal}}(k) + \overline{\mathsf{MultByBit}}(k)) + \overline{\mathsf{Csort}}(n,k)$

Table 6.3: Set operations

try to maintain the set format for sparse columns and the bit format for dense columns simultaneously. The main question is how to find the intersection of a set-represented and a bit-represented column.

On each iteration, the algorithm should now decide, to which columns it applies the set-based approach, and to which the bit-based approach. The table has size $m \times n$ as before. The algorithms convert this set of pairs to set and bit based columns based on the value of m', which defines a sparse column.

We will further assume that all the algorithms are based on xor sharing, as the subalgorithms that we will use are significantly less efficient for additive sharing.

- **Converting a bit matrix column to a set matrix column** (Bits2Set) This algorithm transforms a column of a bit matrix to a column of xor shared row identifiers of length m' where m' is a known upper bound on the number of nonzero entries. This is shown in Alg. 10

  Computing the multiplications is free since we are multiplying by a public value j, and $b_i \in \{0, 1\}$. The only thing that remains is Csort. This transformation is itself already more expensive than multiplying bit vectors, and hence it should be used only if the set representation will be reused afterwards.

- **Converting a set matrix column to a bit matrix column** (Set2Bits) This algorithm is based on finding the characteristic vector of each set element and summing them up. This is shown in Alg. 11.

  The complexity of this algorithm is $m' \odot \overline{\mathsf{CharVec}}(k)$.

Data: A xor shared bit vector $\langle\langle \mathbf{b} \rangle\rangle$ of length m with at most m' nonzero entries
Result: A xor shared set representation $\langle\langle \mathbf{c} \rangle\rangle$ of $\langle\langle \mathbf{b} \rangle\rangle$

foreach $i \in \{1, \ldots, m\}$ **do**
  $\langle\langle \mathbf{c}_i \rangle\rangle = \langle\langle \mathbf{b}_i \rangle\rangle \cdot i$;
end
$\langle\langle \mathbf{d} \rangle\rangle = \mathsf{Csort}(\langle\langle \mathbf{b} \rangle\rangle \bowtie \langle\langle \mathbf{c} \rangle\rangle)$;
return $\langle\langle \mathbf{d} \rangle\rangle[0 : m', 1]$;

Algorithm 10: Bit vector to a set Bits2Set

Data: A xor shared set representation $\langle\langle \mathbf{c} \rangle\rangle$ of length m' with over m elements
Result: A xor shared bit vector representation $\langle\langle \mathbf{b} \rangle\rangle$ of $\langle\langle \mathbf{c} \rangle\rangle$
1 **foreach** $i \in \{1, \ldots, m'\}$ **do**
2 $\quad \langle\langle d_i \rangle\rangle = \mathsf{CharVec}(\langle\langle c_i \rangle\rangle, m)$;
end
3 $\langle\langle \mathbf{b} \rangle\rangle = \bigoplus_{i=1}^{m} \langle\langle d_i \rangle\rangle$;
4 **return** $\langle\langle \mathbf{b} \rangle\rangle$;

Algorithm 11: Set to a bit vector Set2Bits

Type	Operation	Secure Operation Complexity
xor	$\mathsf{Sets2Bits}(m', k)$	$m' \odot \overline{\mathsf{CharVec}}(k)$
	$\mathsf{Bits2Set}(m, k)$	$\overline{\mathsf{Csort}}(m, k)$

Table 6.4: Complexities of bit-set conversion protocols

The summary of the auxiliary protocols is given in Tab. 6.4.

6.4.5 Parallelization issues

One important advantage of bit approach is that if we need to find the intersection of the same column with several other columns, then we can apply OuterProd instead of ordinary multiplication, which is much more efficient. Hence it is not very fair to compare the two approaches by the cost of one intersection. Although it is not applicable to Eclat and Diffset, it can still be found in Apriori and optimized hybrid Apriori-Eclat algorithms that attempt to partially parallelize the search of Eclat, as it was done in [4]. More discussion on this topic can be found in [29].

6.4.6 Other possible optimizations

There are some places that can be optimized, but whose theoretical efficiency has not been estimated yet.

- Our intersection and set difference algorithms for sparse columns provide sorted output. Converting a bit vector to a set vector also provides sorted output, and there are no other ways of obtaining set represented columns. Hence we may assume that both inputs of the intersection and set difference algorithms are already sorted, but the radix sort and the quick sort do not make any use of this assumption. Instead, we could apply just a single step of merged sort that takes two sorted arrays as an input and merges them together. For this, we could use for example the sorting network approach.

- When we are finding an intersection of a set represented column with a bit represented column, we first need to transform the set represented column to a bit column, and then transform the result back to the set representation since it is definitely sparse. Instead, we could apply some kind of oblivious read directly, treating set elements as private indices, and the bit vector as the array from which the values are read. Here we could use the private lookup protocol that we reported in deliverable D2.2.2 [5, Chapter 4].

Chapter 7

Business Process Engineering and Secure Multiparty Computation

A business process is a collection of structured activities and events that allows an enterprise to achieve a business goal. Business processes are operated by business functional units whose tasks are either performed manually or are computer-based. Unambiguous specification of the enterprise's business processes is necessary to monitor the process performance, to automate the computer aided tasks and to re-engineer the enterprise structure. For this reason several high level modeling languages have been proposed (e.g. BPMN [45] and EPC [42]). There is a general agreement (see, e.g. [41]) that well-formed business processes correspond to bounded Petri nets (or more specifically, sound workflow nets) and service automata [31], and several proposals (e.g. [14]) demonstrate techniques to convert high-level models (such as BPMN) to Petri nets or service automata.

Business processes often involve the structuring of activities of several organizations. This is the case when several potentially competitive enterprises can share their knowledge and skills to form a temporary alliance, usually called a virtual enterprise (VE), in order to catch new business opportunities. Virtual enterprises can be part of long-term strategic alliances or short-term collaborations. To effectively manage a virtual enterprise, receiving well-founded support from business process engineering techniques is critical.

One of the main obstacles to such business process engineering is the perceived threat to the participants' autonomy. In particular, the participants can be reluctant to expose their internal processes or logs, as this knowledge can be analyzed by the other participants to reveal sensitive information such as efficiency secrets or weaknesses in responding to market demand. Moreover, the value of confidentiality of business processes is widely recognized.

In this Chapter we use SMC techniques to handle two problems in this context: *VE process fusion*, *Business Process Matching* and *Log auditing*.

VE process fusion. The dependencies among the activities of a prospective VE cross the boundaries of the VE constituents. It is, therefore, crucial to allow the VE constituents to discover their local views of the inter-organizational workflow, enabling each company to re-shape, optimize and analyze the possible local flows that are consistent with the processes of the other VE constituents. We refer to this problem as *VE process fusion*. Even if it has been widely investigated, no previous work addresses VE process fusion in the presence of privacy constraints.

Business Process Matching. A non-trivial process engineering task is to ensure soundness (i.e., interoperability) of the collaboration of VE constituents: can the interactions between the partners lead to a deadlock; are the involved parties guaranteed to terminate properly; can the collaboration lead to documents that are never collected by the recipient? *Business process matching* is the activity of checking whether a given business process can interoperate with another one in a correct manner. In case the check fails, it is desirable to obtain information about how the first process can be corrected with as few modifications as

possible to achieve interoperability. In case the two business processes belong to two separate enterprises that want to build a virtual enterprise, business process matching based on revealing the business processes poses a clear threat to privacy, as it may expose sensitive information about the inner operation of the enterprises. We propose a measure for similarity between business processes and use this measure to devise an algorithm that constructs the most similar process to the first one that can interoperate with the second one.

Log auditing. This problem consists in enabling a participant that owns a business process to check if the partner's logs match its business process, thus enabling the two partners to discover failures, errors and inefficiencies.

7.1 Virtual Enterprise Process Fusion

In [19] we investigate a mechanism to establish cross-organizational business processes, or more precisely, to identify for each participant of a virtual enterprise (VE) which operations can be performed locally. In other words, we need to compute the contributing subset of the existing local business process that is consistent with the processes of the other VE constituents. We refer to this problem as *VE process fusion*.

Here we consider two mutually distrustful parties, each following a local business process, that wish to compute their local view of the VE process, assuming no trusted third party is available. The VE process is modeled as the synchronous composition of the participant work-flows, and each participant's local view is represented by a process that is trace equivalent to the VE process up to transitions that are not observable by the participant itself. The two parties are reluctant to reveal any information about their own business process that is not strictly deducible from the local view of the other party.

Assume two enterprises a and b, with their own business processes, that cooperate to build a VE. For each of the two enterprises we are given a local alphabet, Σ_a and Σ_b, respectively. Each enterprise also owns a local business process, representing all possible allowed executions, that is given as a bounded labelled Petri net (N_a and N_b, respectively) that is defined over the corresponding local alphabet. For example, consider the Petri nets depicted in Fig. 7.1a and 7.1b. The symbols of the alphabets can represent various types of actions or events:

1. an internal task of the enterprises (the boxes labelled E, D, G and H, standing for tasks such as the packaging of goods and the like),

2. an interaction between the two enterprises (A and B, representing tasks such as the exchange of electronic documents),

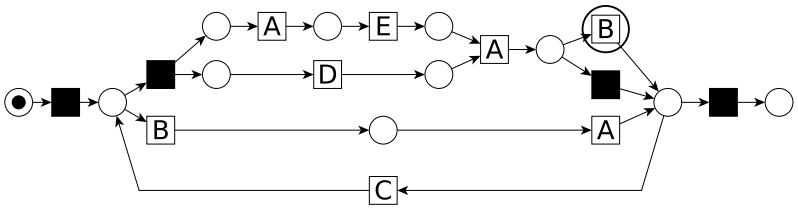
3. an event observed by one of the enterprises only (P, the receipt of a payment),

4. an event observed by both enterprises (C, the departure of a carrier from the harbor), and

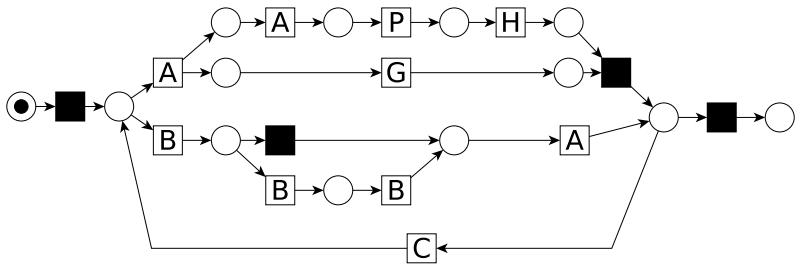
5. a silent event (black boxes, usually used to simplify net structure).

Let N be a labeled Petri net defined over the alphabet Σ. We use $\mathbf{proj}_{\Sigma'}(N)$ to represent the *projection* of the net on an alphabet $\Sigma' \subseteq \Sigma$, obtained by hiding all transitions that are labeled with symbols not contained in Σ'. With each execution of the net, we associate a *trace*, which is a word in Σ^*. The set of all traces of the net is called the *language* of N and is denoted by $\mathcal{L}(N)$.

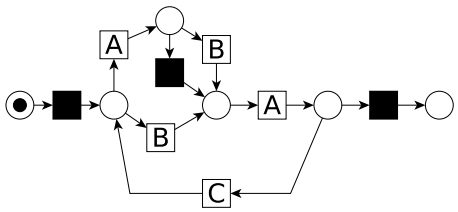
Let N_1 and N_2 be two labeled Petri nets defined over the alphabets Σ_1 and Σ_2 respectively. The two nets are *trace equivalent*, denoted $N_1 \sim N_2$, whenever $\mathcal{L}(N_1) = \mathcal{L}(N_2)$. Finally, the *product* of the two nets $N_1 \times N_2$ is a labeled Petri net (over the alphabet $\Sigma_1 \cup \Sigma_2$) that synchronizes the transitions of the original two nets that carry the same label. The Petri net product offers a natural way to model complex systems by composing elementary components.



(a) N_a



(b) N_b



(c) N'_b

Figure 7.1: Business processes modeled with Petri nets

The problem of *VE process fusion* can be defined as computing the mapping:

$$N_i \mapsto N'_i \qquad (i \in \{a, b\})$$

where $N'_i \sim \mathbf{proj}_{\Sigma_i}(N_a \times N_b)$. This describes the problem of each participant computing from its local business process N_i a new process, consistent with the global VE business process represented by the synchronous composition. If there are no privacy constraints, then implementing VE process fusion is straightforward: the two parties can simply (1) exchange their Petri nets, (2) compute the product Petri net $N_a \times N_b$, and finally (3) project $N_a \times N_b$ on their respective alphabet Σ_i.

To illustrate VE process fusion, consider the following running example. Let a and b be two enterprises, with business processes as shown in Fig. 7.1a and 7.1b, respectively. Intuitively, when enterprise a is fused with enterprise b, its business process must be updated so as to satisfy the partner's constraints. For instance, an analysis of the fusion suggested above will reveal that the encircled activity B in Fig. 7.1a cannot be executed any more after the fusion.

Here we are interested in preserving the participants' privacy. In particular, we wish the two participants to obtain N'_a and N'_b, respectively, without being able to learn about the other enterprise's processes more than what can be deduced from their own process (i.e. the *private input*) and the obtained result (i.e. the *private output*). Apart from the processes, we also consider the alphabet differences to be private. That is, we consider public just the common alphabet $\Sigma_a \cap \Sigma_b$ (i.e. the events of type 2 and 4). For example, regardless whether enterprise b owns the business process N_b or N'_b from Fig. 7.1, the sub-process of N_a that is consistent with the observable partner's constraints is one and the same, namely the Petri net shown in Fig. 7.2 (presented for convenience as the superimposition of the observable external constraints to the original process of enterprise a). Therefore, the mechanism used to implement process fusion should not allow party a to distinguish between N_b and N'_b or any other partner process that gives rise to the same local view of a.

We address VE process fusion by lifting the problem to privately intersect regular languages. In [20] we addresses the problem of computing the intersection of regular languages in a privacy-preserving fashion. Private set intersection has been addressed earlier in the literature, but for finite sets only. We discuss the various possibilities for solving the problem efficiently, and argue for an approach based on minimal deterministic finite automata (DFA) as a suitable, non-leaking representation of regular language intersection. We propose two different algorithms for DFA minimization in a secure multiparty computation setting,

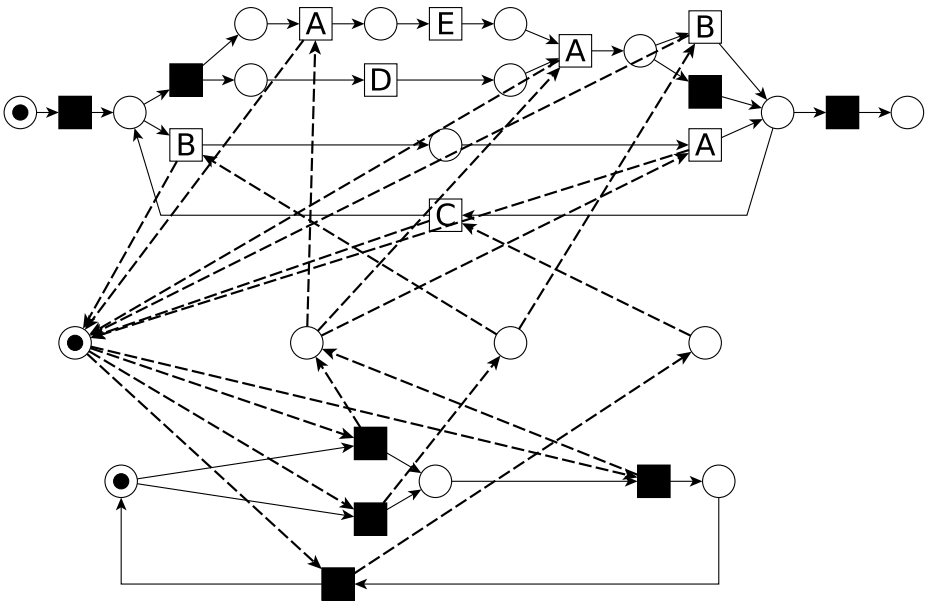


Figure 7.2: The local view of a after its fusion with b

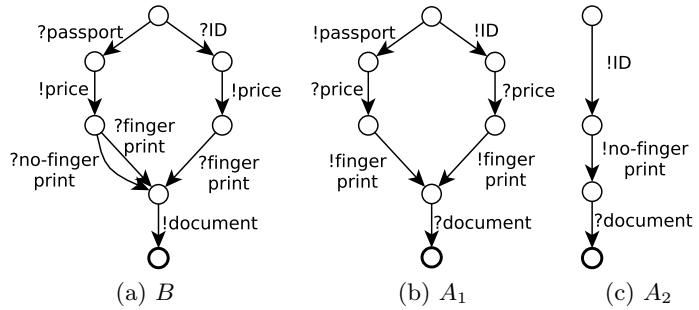
illustrating different aspects of programming based on universal composability and the constraints this sets on existing algorithms. The implementation of our algorithms is based on the programming language SecreC. In [19] we demonstrate how these results can be extended to deal with business processes that are formally modeled with bounded Petri nets, and thus addressing VE process fusion. Furthermore, we provide a prototype implementation of our proposed technique, developed as a plug-in of ProM [44], a well known business process analysis platform.

7.2 Privacy Preserving Business Process Matching

A non-trivial process engineering task is to ensure soundness (i.e., interoperability) of enterprise collaborations. Whenever applicable, top-down development approaches such as [43, 22] can be used to ensure soundness of the interorganizational processes, while meeting privacy requirements. Such approaches typically consist of three phases: (i) the participants agree upon a global (and publicly known) process that is subject to global soundness analyses, (ii) from the global process, local interfaces (one for each participant) are automatically synthesized, and (iii) each partner locally implements a private process and checks that it conforms to its public interface. This scheme allows to transfer the soundness properties of the global process to the composition of the private processes *without* publishing the latter.

If the agreement phase is not desirable, and the corresponding definition of a shared business process is not possible, a bottom-up approach must be taken instead. Each participant owns then an already existing private process, and the process engineering task becomes the one of guaranteeing that their composition is sound. Such a bottom-up approach must face two problems: (i) the global process and the public interfaces are not available, thus the soundness of the composition of "secret" processes must be checked without revealing information about the constituents, and (ii) if the soundness check fails, a Boolean result is of little use, since the participants have no information about the global process and thus can not investigate what is wrong. This prevents the participants from adapting their local procedures to form a sound collaboration.

In [21] we present a bottom-up approach to checking soundness of interorganizational business processes that addresses both these problems. We demonstrate a scenario based on an example from [43], assuming two parties p_A and p_B that own private business processes represented as service automata A and B. Figure 7.3a depicts the service automaton of a registration office. According with a customer request, this office can prepare passports and ID cards. In both cases, the office informs the customer about the cost and, if



(a) B (b) A_1 (c) A_2

Figure 7.3: A registration office and two customer processes

needed, stores the customer's fingerprints. Finally, it delivers the requested document to the customer. Two customers (Figures 7.3b and 7.3c) check whether they can collaborate with this office. Assuming soundness means deadlock freedom, the soundness check will succeed for customer A_1 and fail for customer A_2. In fact, the second customer requests an ID card without providing the necessary fingerprints, thus leading the collaboration to a deadlock, and it never collects the delivered invoice.

To address problem (ii) above, in [21] we introduce a measure for *behavioral similarity* between two automata. Then, *Business Process Matching* is the problem of checking whether the composition of the automata A and B is sound, and in case it is not, to allow the participant p_A to discover, among all automata that can be soundly composed with B, the one that is most similar to A. This enables a correction loop, where one of the two participants can refine its own process (possibly accepting the suggested correction) and repeat the soundness check. This goal is achieved under strong privacy constraints: nothing more is leaked to the participants or any third party than what can be deduced from the result.

Our approach is based on the combination of three techniques: (i) we lift the check of the soundness of the automata composition to matching one of the automata against the *operating guideline* of the other (i.e., a representation of all service automata that can soundly collaborate with it, see [31]), (ii) we introduce the notion of *weighted matching* as a measure for matching degree, and show how to compute this measure and in the same time extract the behaviourally most similar service that can soundly collaborate with the other party, and (iii) we implement an algorithm to compute weighted matching while preserving privacy by means of Secure Multiparty Computation (SMC) techniques.

Our proposal is a bottom-up mechanism to match business processes and to suggest suitable corrections. Existing approaches for process matching that take into account secrecy of the input processes are top-down and require to disclose the complete set of strategies of the participants (by either publishing a public interface [43] or by disclosing the complete operating guideline). We go beyond these results, by only leaking one of the possible strategies for the partner's process.

7.3 Log Auditing

Identifying errors in activities that involve several business partners and cross enterprise boundaries is challenging, because these activities must respect the policies of all involved partners. From a behavioral point of view, such constraints can be rephrased to require that all process instances are consistent with the business processes of all partners.

Log auditing consists of checking whether the log of a terminated process instance matches a predefined business process, and of identifying possible mismatches. Here we consider two mutually distrustful parties, one party owning the log and the other one owning the business process, assuming no trusted third party is available. The two parties are reluctant to reveal any information about their log and business process that is not strictly deducible from the matching result.

Assume two enterprises a and b. For each of the two enterprises we are given local alphabets, Σ_a and Σ_b, respectively. Similarly to Section 7.1, we assume that $\Sigma = \Sigma_a \cap \Sigma_b$ are events and interactions shared

between the two enterprises, while $(\Sigma_a \cup \Sigma_b) \setminus \Sigma$ are internal tasks. Enterprise a recorded the log of a process instance by monitoring enterprise activities; the log is given as a word $\omega \in \Sigma_a^*$. Enterprise b owns a local business process, representing all possible licit executions, that is given as a bounded labelled Petri net N_b defined over the corresponding local alphabet.

Formally, the problem of *log auditing* is to compute whether

$$\mathbf{proj}_{\Sigma_a \cap \Sigma_b}(\omega) \in \mathbf{proj}_{\Sigma_a \cap \Sigma_b}(\mathcal{L}_b) \ .$$

If there are no privacy constraints, then log auditing is simple: the two parties can simply exchange their logs and processes, and replay the string ω on the Petri net.

In [27], we enable the two participants to check the matching without being able to learn about the other enterprise's process or log more than what can be deduced from the own input and the obtained result. Apart from the process and the log, we also consider the alphabet differences as private values. Hence, we consider only the common alphabet $\Sigma_a \cap \Sigma_b$ as a public value. To execute log auditing without compromising the participants' privacy we introduce a new SMC protocol for private lookup.

Bibliography

[1] Baruch Awerbuch and Yossi Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Computers*, 36(10):1258–1263, 1987.

[2] Dan Bogdanov, Yiannis Giannakopoulos, Roberto Guanciale, Liina Kamm, Peeter Laud, Pille Pruulmann-Vengerfeldt, Riivo Talviste, Kadri Tõldsepp, and Jan Willemson. Scientific Progress Analysis and Recommendations, January 2013. UaESMC Deliverable 5.2.1.

[3] Dan Bogdanov, Roberto Guanciale, Liina Kamm, Peeter Laud, Riivo Talviste, and Jan Willemson. Advances in SMC techniques, January 2013. UaESMC Deliverable 2.2.1.

[4] Dan Bogdanov, Roman Jagomägis, and Sven Laur. A universal toolkit for cryptographically secure privacy-preserving data mining. In *Proceedings of the 2012 Pacific Asia Conference on Intelligence and Security Informatics*, PAISI'12, pages 112–126, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Dan Bogdanov, Liina Kamm, Peeter Laud, Alisa Pankova, Pille Pullonen, Riivo Talviste, and Jan Willemson. Advances in SMC techniques, January 2014. UaESMC Deliverable 2.2.2.

[6] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512, 2014. `http://eprint.iacr.org/`.

[7] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-polymorphic programming of privacy-preserving applications. In Alejandro Russo and Omer Tripp, editors, *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014*, page 53. ACM, 2014.

[8] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[9] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.

[10] Xiang Cheng, Sen Su, Shengzhi Xu, and Zhengyi Li. Dp-apriori: A differentially private frequent itemset mining algorithm based on transaction splitting. *Computers & Security*, 50:74–90, 2015.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 23.2 The algorithms of Kruskal and Prim, pages 567–574. MIT Press and McGraw-Hill, 2nd edition, 2001.

[12] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[13] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. The Internet Society, 2015.

[14] Remco M Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, 2008.

[15] Cynthia Dwork. A firm foundation for private data analysis. *Commun. ACM*, 54(1):86–95, 2011.

[16] Hamid Ebadi, David Sands, and Gerardo Schneider. Differential privacy: Now it's getting personal. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 69–81. ACM, 2015.

[17] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

[18] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.

[19] Roberto Guanciale and Dilian Gurov. Privacy preserving business process fusion. In Fabiana Fournier and Jan Mendling, editors, *Business Process Management Workshops - BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers*, volume 202 of *Lecture Notes in Business Information Processing*, pages 96–101. Springer, 2014.

[20] Roberto Guanciale, Dilian Gurov, and Peeter Laud. Private Intersection of Regular Languages. In *Proceedings of the 12th Annual Conference on Privacy, Security and Trust*, pages 112–120. IEEE, 2014.

[21] Dilian Gurov, Peeter Laud, and Roberto Guanciale. Privacy Preserving Business Process Matching. In *Proceedings of the 13th Annual Conference on Privacy, Security and Trust*. IEEE, 2015.

[22] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008.

[23] Justin Hsu, Marco Gaboardi, Andreas Haeberlen, Sanjeev Khanna, Arjun Narayan, Benjamin C. Pierce, and Aaron Roth. Differential privacy: An economic method for choosing epsilon. In Anupam Datta and Cedric Fournet, editors, *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 398–410. IEEE, 2014.

[24] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[25] Liina Kamm, Peeter Laud, Alisa Pankova, and Pille Pullonen. Advances in SMC techniques, July 2015. UaESMC Deliverable 2.2.3.

[26] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[27] Peeter Laud. A private lookup protocol with low online complexity for secure multiparty computation. In Siu Ming Yiu and Elaine Shi, editors, *ICICS 2014*, LNCS. Springer, 2014. To appear.

[28] Peeter Laud. Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees. *Proceedings of Privacy Enhancing Technologies*, 2015(2):188–205, 2015.

[29] Peeter Laud and Alisa Pankova. Privacy-preserving frequent itemset mining for sparse and dense data. Cryptology ePrint Archive, Report 2015/671, 2015. `http://eprint.iacr.org/`.

[30] Jaewoo Lee and Christopher W. Clifton. Top-k frequent itemsets via differentially private fp-trees. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 931–940, New York, NY, USA, 2014. ACM.

[31] Niels Lohmann, Peter Massuthe, and Karsten Wolf. Operating guidelines for finite-state services. In *Proceedings of ICATPN'07*, volume 4546 of *Lecture Notes in Computer Science*, pages 321–341. Springer, 2007.

[32] Ashwin Machanavajjhala and Daniel Kifer. Designing statistical privacy for your data. *Commun. ACM*, 58(3):58–67, 2015.

[33] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David E. Culler. GUPT: privacy preserving data analysis made easy. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 349–360. ACM, 2012.

[34] Jaroslav Nešetřil, Eva Milkovà, and Helena Nešetřilovà. Otakar Borůvka on minimum spanning tree problem; Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1-3):3–36, 2001.

[35] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 75–84. ACM, 2007.

[36] Martin Pettai and Peeter Laud. Combining Differential Privacy and Secure Multiparty Computation. Cryptology ePrint Archive, Report 2015/598, 2015. `http://eprint.iacr.org/`.

[37] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[38] Sander Siim. Privacy-Preserving String Matching with PRAM Algorithms. Cryptography Seminar report, University of Tartu, December 2014. `https://courses.cs.ut.ee/MTAT.07.022/2014_fall/uploads/Main/sander-report-f14.pdf`.

[39] Adam Smith. Privacy-preserving statistical estimation with optimal convergence rates. In Lance Fortnow and Salil P. Vadhan, editors, *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 813–822. ACM, 2011.

[40] Chongjing Sun, Yan Fu, Junlin Zhou, and Hui Gao. Personalized privacy-preserving frequent itemset mining using randomized response. *The Scientific World Journal*, 2014, 2014.

[41] Wil MP van der Aalst. The application of Petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66, 1998.

[42] Wil MP van der Aalst. Formalization and verification of event-driven process chains. *Information and Software technology*, 41(10):639–650, 1999.

[43] Wil MP van der Aalst, Niels Lohmann, Peter Massuthe, Christian Stahl, and Karsten Wolf. Multiparty contracts: Agreeing and implementing interorganizational processes. *The Computer Journal*, 53(1):90–106, 2010.

[44] Boudewijn F van Dongen, Ana Karla A de Medeiros, HMW Verbeek, AJMM Weijters, and Wil MP van der Aalst. The prom framework: A new era in process mining tool support. In *Applications and Theory of Petri Nets*, pages 444–454. Springer, 2005.

[45] Stephen A White. Introduction to BPMN. *IBM Cooperation*, 2, 2004.

[46] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.

[47] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 326–335, New York, NY, USA, 2003. ACM.

[48] Chen Zeng, Jeffrey F. Naughton, and Jin-Yi Cai. On differentially private frequent itemset mining. *Proc. VLDB Endow.*, 6(1):25–36, November 2012.