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 D2.2.2
Advances in Secure Multiparty Protocols

Due date of deliverable: 31st January 2014

Actual submission date: 31st January 2014

Start date of the project: **1st February 2012** Duration: **36 months**

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

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:
Advances in Secure Multiparty Protocols

This document summarizes deliverable D2.2.2 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 contains an overview of the results of the second year of UaESMC, pertaining to secure multiparty computation techniques. The studies of these techniques have been directed by the example problems selected during the first year, as well as by the desire to have a comprehensive framework of privacy-preserving computation techniques by the end of the project.

In this deliverable, we report of the following findings and advances:

- We provide improved privacy preserving algorithms for giving an overview of the data. We also give privacy preserving versions of algorithms for several most common statistical tests. As result, we were able to conduct a full-scale experimental statistical study so that confidential data were always processed using SMC. The strengths of our solution are generality, precision and practicality. We show that secure multi-party computation is flexible enough for implementing complex applications.
- We have found that the class of techniques currently used for problem transformation based solving of linear programming tasks cannot be privacy-preserving. This leaves the implementations of standard LP-solving algorithms on top of generic protocol sets for privacy-preserving arithmetic as the only general method for privacy-preserving LP, unless some radically new ideas for transforming LP problems are proposed.
- We provide efficient algorithms for privacy-preserving finite automata execution, that achieve online efficiency through offline precomputations.
- We provide a protocol set for actively-secure two-party computation that also acheives efficiency through offline precomputations.
- We provide a protocol transformation that turns any passively secure multiparty computation protocol with honest majority to a protocol where any misbehaviour is detected after the execution.

We expect many of these advances to play a significant role in the UaESMC framework.

List of Authors

Dan Bogdanov (CYB)   Liina Kamm (CYB)   Peeter Laud (CYB)   Alisa Pankova (CYB)
Pille Pullonen (CYB)   Riivo Talviste (CYB)   Jan Willemson (CYB)

Contents

Chapter 1

Introduction

This report gives a review of the advances in secure multiparty computation techniques made during the second year of the UaESMC project. We have investigated several different tasks and problems, making progress in solving particular classes of computational problems, as well as in improving the security guarantees of broad classes of protocols. Our investigations have been motivated by the example problems selected during the first year [7]. Even more, they have been motivated by the desire to have a comprehensive set of SMC techniques available for the UaESMC framework, due to be formulated during the final year of the project.

We have continued our work on statistical analysis of structured data, the results of which are reported in Chapter 2. The processing of structured data requires efficient database operations, which in turn depend on fast sorting methods. We have thus performed a thorough effeciency comparison of privacy-preserving sorting methods, described in Chapter 7. In Chapter 3, we describe our surprising results on linear programming, obtained during this year of UaESMC. Both statistical analysis and linear programming are among the selected example problems of UaESMC.

There is a different kind of problem that we have also investigated during the second year of UaESMC project. It pertains to the privacy-preserving execution of finite automata. This problem has applications in network management (also a selected example problem of UaESMC). It is also interesting because the access patterns for the algorithms solving it significantly depend on the data that we would like to remain private. Efficient privacy-preserving solutions for this problem would thus need new kinds of techniques. Our results are reported in Chapter 4.

We have investigated efficient methods to make the parties of a privacy-preserving computation or, more generally, any cryptographic protocol faithfully perform the instructions of the protocol. In Chapter 6 we describe a protocol set for actively secure multiparty computation among two parties. The efficiency of the protocols is achieved through offline precomputation. In Chapter 5, we show how to turn any protocol secure against semi-honest adversaries into a protocol secure against covert adversaries (such adversaries may deviate from the protocol, but only if they are not caught afterwards), under the condition that a majority of protocol participants are honest. Both techniques allow secure computation application to achieve stronger security properties.

To the end of this deliverable, we have annexed a number of papers and technical reports we have published during the second year of the project. These papers are referenced from the main body of the deliverable.

Chapter 2

Privacy-Preserving Statistical Analysis

2.1 Simple Statistics

This year, we continued our research into privacy-preserving statistical analysis. We finished and improved work begun last year and we added more statistical tests to the statistics suite in order to provide a wider choice for data analysts. In the following, let $[\![x]\!]$ denote a private value x, let $[\![\vec{a}]\!]$ denote a private value vector $\vec{a}$, and let binary operations between vectors be point-wise operations.

2.1.1 Quantiles and Outlier Detection

The first improvement to deliverable D2.2.1 [9] is the new quantile calculation method. As no one method for computing quantiles has been widely agreed upon in the statistics community, we use algorithm $\mathbf{Q_7}$ from [31], because it is the default choice in our reference statistical analysis package GNU R. Let p be the percentile we want to find and let $[\![\vec{a}]\!]$ be a vector of values sorted in ascending order. Then the quantile is computed using the following function:

$$\mathbf{Q}_7(p, [\![\vec{a}]\!]) = (1-\gamma) \cdot [\![\vec{a}]\!][j] + \gamma \cdot [\![\vec{a}]\!][j+1] \ ,$$

where $j = \lfloor (n-1)p \rfloor + 1$, n is the size of vector $[\![\vec{a}]\!]$, and $\gamma = np - \lfloor (n-1)p \rfloor - p$. Once we have the index of the quantile value, we can use oblivious versions of vector lookup or sorting to learn the quantile value from the input vector.

While data-independent oblivious sorting can easily be implemented using sorting networks, oblivious Hoare's selection is more complex, because the partitioning sub-procedure publishes random comparison results the same way as cutting does. We solve the problem in the same way, by running a shuffling procedure before the selection. As the elements of the resulting vector are in random order, even the declassification of all comparison results leaks no information about the input vector. Hence, it is straightforward to simulate the outcome of the entire selection algorithm. As Hoare's selection algorithm has linear asymptotic complexity whereas common sorting networks consist of $\Theta(n \log^2 n)$ comparison gates, selection is potentially faster[1] if we are dealing with large datasets. Although we implemented both approaches, we did not observe this in practice. In fact, using sorting networks turned out to be faster, so we chose this as our default solution.

We also implemented a very simple outlier elimination method using quantiles. We do not need to publish the quantile to use it for outlier filtering. Let q_0 and q_1 be the 5% and 95% quantiles of an attribute $[\![\vec{a}]\!]$. It is common to mark all values smaller than q_0 and larger than q_1 as outliers. The corresponding mask vector is computed by comparing all elements of $[\![\vec{a}]\!]$ to $\mathbf{Q_7}(0.05, [\![\vec{a}]\!])$ and $\mathbf{Q_7}(0.95, [\![\vec{a}]\!])$, and then multiplying the resulting index vectors.

[1]As the asymptotic complexity of shuffle is $\Theta(n \log n)$, which is the complexity of the optimal AKS sorting network, both approaches are theoretically equivalent.

Algorithm 1: Function **cut** for cutting the dataset according to a given filter.

```
Data: Data vector $[\![\vec{a}]\!]$ of size $N$ and corresponding mask vector $[\![\vec{m}]\!]$.
Result: Data vector $[\![\vec{x}]\!]$ of size $n$ that contains only elements of $[\![\vec{a}]\!]$ corresponding to the mask $[\![\vec{m}]\!]$
1 Obliviously shuffle the value pairs in vectors $([\![\vec{a}]\!], [\![\vec{m}]\!])$ into $([\![\vec{a'}]\!], [\![\vec{m'}]\!])$
2 $\vec{s} \leftarrow$ publish$([\![\vec{m'}]\!])$
3 $[\![\vec{x}]\!] \leftarrow ([\![\vec{a'}]\!][i] \mid \vec{s}[i] = 1, i \in \{1, \ldots, N\})$
4 return $[\![\vec{x}]\!]$
```

Algorithm 2: Algorithm for finding the five-number summary of a vector.

```
Data: Input data vector $[\![\vec{a}]\!]$ and corresponding mask vector $[\![\vec{m}]\!]$.
Result: Minimum $[\![min]\!]$, lower quartile $[\![lq]\!]$, median $[\![me]\!]$, upper quartile $[\![uq]\!]$, and maximum
        $[\![max]\!]$ of $[\![\vec{a}]\!]$ based on the mask vector $[\![\vec{m}]\!]$
1 $[\![\vec{x}]\!] \leftarrow$ cut$([\![\vec{a}]\!], [\![\vec{m}]\!])$
2 $[\![\vec{b}]\!] \leftarrow$ sort$([\![\vec{x}]\!])$
3 $[\![min]\!] \leftarrow [\![\vec{b}]\!][1]$
4 $[\![max]\!] \leftarrow [\![\vec{b}]\!][n]$
5 $[\![lq]\!] \leftarrow \mathbf{Q_7}(0.25, [\![\vec{b}]\!])$
6 $[\![me]\!] \leftarrow \mathbf{Q_7}(0.5, [\![\vec{b}]\!])$
7 $[\![uq]\!] \leftarrow \mathbf{Q_7}(0.75, [\![\vec{b}]\!])$
8 return $([\![min]\!], [\![lq]\!], [\![me]\!], [\![uq]\!], [\![max]\!])$
```

2.1.2 Five-Number Summary and Frequency Tables

It is important for a data analyst to get an overview of the data. As it is not possible to see the data in SMC format, we give an overview using the five number summary and the histogram.

First, for reference, we give Algorithm 1 for obliviously cutting the dataset based on a given filter. First the value and mask vector pairs are obliviously shuffled, retaining the correspondence of the elements. Next, the mask vector is declassified and values for which the mask vector contains 0 are removed from the value vector. The obtained cut vector is then returned to the user. This process leaks the number of values that correspond to the filters that the mask vector represents. This makes cutting trivially safe to use, when the number of records in the filter would be published anyway. Oblivious shuffling ensures that no other information about the private input vector and mask vector is leaked [36]. Therefore, all algorithms that use oblivious cut provide source privacy.

Algorithm 2 describes the computation of the five-number summary of a value vector $[\![\vec{a}]\!]$ with the corresponding mask vector $[\![\vec{m}]\!]$. Function **cut** leaks the count of elements n that correspond to the filter signified by the mask vector $[\![\vec{m}]\!]$. However, the filter size is often one of the descriptive statistics that analysts want to learn. If we want to keep n secret, we can use Algorithm 3. This hides n, but runs slower than Algorithm 2.

More information about the data can be obtained by looking at the distribution of a data attribute. For categorical attributes, this can be done by computing the frequency of the occurrences of different values. For numerical attributes, we must split the range into bins specified by breaks and compute the corresponding frequencies. The resulting frequency table can be visualised as a histogram. The algorithm publishes the number of bins and the number of values in each bin. We also implemented the histogram calculation algorithm.

Algorithm 4 computes a frequency table for a vector of values similarly to a public frequency calculation algorithm.

Algorithm 3: Oblivious algorithm for finding the five-number summary of a vector.

```
Data: Input data vector $[\![\vec{a}]\!]$ and corresponding mask vector $[\![\vec{m}]\!]$.
Result: Minimum $[\![min]\!]$, lower quartile $[\![lq]\!]$, median $[\![me]\!]$, upper quartile $[\![uq]\!]$, and maximum
        $[\![max]\!]$ of $[\![\vec{a}]\!]$ based on the mask vector $[\![\vec{m}]\!]$
1 $([\![\vec{b}]\!], [\![\vec{m'}]\!]) \leftarrow \mathbf{sort}([\![\vec{a}]\!], [\![\vec{m}]\!])$
2 $[\![n]\!] \leftarrow \mathbf{sum}([\![\vec{m}]\!])$
3 $[\![os]\!] \leftarrow [\![N - n]\!]$
4 $[\![min]\!] \leftarrow [\![\vec{b}]\!][1 + [\![os]\!]]$
5 $[\![max]\!] \leftarrow [\![\vec{b}]\!][N]$
6 $[\![lq]\!] \leftarrow \mathbf{Q_7}(0.25, [\![\vec{a}]\!], [\![os]\!])$
7 $[\![me]\!] \leftarrow \mathbf{Q_7}(0.5, [\![\vec{a}]\!], [\![os]\!])$
8 $[\![uq]\!] \leftarrow \mathbf{Q_7}(0.75, [\![\vec{a}]\!], [\![os]\!])$
9 return $([\![min]\!], [\![lq]\!], [\![me]\!], [\![uq]\!], [\![max]\!])$
```

Algorithm 4: Algorithm for finding the frequency table of a data vector.

```
Data: Input data vector $[\![\vec{a}]\!]$ and corresponding mask vector $[\![\vec{m}]\!]$.
Result: Vector $[\![\vec{b}]\!]$ containing breaks against which frequency is computed, and vector $[\![\vec{c}]\!]$ containing
        counts of elements
1 $[\![\vec{x}]\!] \leftarrow \mathbf{cut}([\![\vec{a}]\!], [\![\vec{m}]\!])$
2 $n \leftarrow \mathbf{count}([\![\vec{x}]\!])$
3 $k \leftarrow \lceil \log_2(n) + 1 \rceil$
4 $[\![min]\!] \leftarrow \mathbf{min}([\![\vec{x}]\!])$, $[\![max]\!] \leftarrow \mathbf{max}([\![\vec{x}]\!])$
5 Compute breaks according to $[\![min]\!]$, $[\![max]\!]$ and $k$, assign result to $[\![\vec{b}]\!]$
6 $[\![\vec{c}]\!][1] = (\mathbf{count}([\![\vec{x}]\!][i]) \mid [\![\vec{b}]\!][1] \leq [\![\vec{x}]\!][i] \leq [\![\vec{b}]\!][2], i = 1, \ldots, n)$
7 $[\![\vec{c}]\!][j] = (\mathbf{count}([\![\vec{x}]\!][i]) \mid [\![\vec{b}]\!][j] < [\![\vec{x}]\!][i] \leq [\![\vec{b}]\!][j+1], i = 1, \ldots, n), j = 2, \ldots, k$
8 return $([\![\vec{b}]\!], [\![\vec{c}]\!])$
```

2.2 Statistical Tests

2.2.1 Wilcoxon Rank Sum Test and Signed Rank Test

In addition to the t-test and the paired t-test, we created privacy preserving algorithms for the Wilcoxon rank sum test and signed rank test. As t-tests are formally applicable only if the distribution of attribute values in case and control groups follows the normal distribution. If this assumption does not hold, it is appropriate to use non-parametric Wilcoxon tests. The Wilcoxon rank sum test [29] works on the assumption that the distribution of data in one group significantly differs from that in the other.

A privacy-preserving version of the rank sum test follows the standard algorithm, but we need to use several tricks to achieve output privacy. Algorithm 5 gives an overview of how we compute the test statistic $[\![w]\!]$ using the Wilcoxon rank sum test.

For this algorithm to work, we need to cut the database similarly to what was done for the five-number summary. But we need the dataset to retain elements from both groups—cases and controls. On line 1, we combine the two input mask vectors into one making sure that the values that appear in both masks as 1 are removed from the analysis. The function **cut** on line 2 differs from its previous usage in that several vectors are cut at once based on the combined filter $[\![\vec{m}]\!]$.

Similarly to Student's paired t-test, the Wilcoxon signed-rank test [48] is a paired difference test. Our version, given in Algorithm 6, takes into account Pratt's correction [29] for when the values are equal and their difference is 0. As with the rank sum test, we do not take into account the ranking of equal values, but this only gives us a more pessimistic test statistic.

Algorithm 5: Wilcoxon rank sum test

```
    Data: Value vector $\llbracket\vec{a}\rrbracket$ and corresponding mask vectors $\llbracket\vec{m_1}\rrbracket$ and $\llbracket\vec{m_2}\rrbracket$
    Result: Test statistic $\llbracket w\rrbracket$
1   $\llbracket\vec{m}\rrbracket \leftarrow \llbracket\vec{m_1}\rrbracket + \llbracket\vec{m_2}\rrbracket - (\llbracket\vec{m_1}\rrbracket \cdot \llbracket\vec{m_2}\rrbracket)$
2   $(\llbracket\vec{x}\rrbracket, \llbracket\vec{m'_1}\rrbracket, \llbracket\vec{m'_2}\rrbracket) \leftarrow \mathbf{cut}((\llbracket\vec{a}\rrbracket, \llbracket\vec{m_1}\rrbracket, \llbracket\vec{m_2}\rrbracket), \llbracket\vec{m}\rrbracket)$
3   $(\llbracket\vec{\hat{x}}\rrbracket, \llbracket\vec{\hat{m_1}}\rrbracket\llbracket\vec{\hat{m_2}}\rrbracket) \leftarrow \mathbf{sort}((\llbracket\vec{x}\rrbracket, \llbracket\vec{m'_1}\rrbracket, \llbracket\vec{m'_2}\rrbracket))$
4   $\llbracket\vec{r}\rrbracket \leftarrow \mathbf{rank}(\llbracket\vec{\hat{x}}\rrbracket)$
5   $\llbracket\vec{r_1}\rrbracket \leftarrow \llbracket r \cdot \vec{\hat{m_1}}\rrbracket$ and $\llbracket\vec{r_2}\rrbracket \leftarrow \llbracket r \cdot \vec{\hat{m_2}}\rrbracket$
6   $\llbracket R_1\rrbracket \leftarrow \mathbf{sum}(\llbracket\vec{r_1}\rrbracket)$ and $\llbracket R_2\rrbracket \leftarrow \mathbf{sum}(\llbracket\vec{r_2}\rrbracket)$
7   $\llbracket n_1\rrbracket \leftarrow \mathbf{sum}(\llbracket\vec{m_1}\rrbracket)$ and $\llbracket n_2\rrbracket \leftarrow \mathbf{sum}(\llbracket\vec{m_2}\rrbracket)$
8   $\llbracket u_1\rrbracket \leftarrow \llbracket R_1\rrbracket - \llbracket\frac{n_1\cdot(n_1+1)}{2}\rrbracket$ and $\llbracket u_2\rrbracket \leftarrow \llbracket n_1 \cdot n_2\rrbracket - \llbracket u_1\rrbracket$
9   return $\llbracket w\rrbracket \leftarrow \mathbf{min}(\llbracket u_1\rrbracket, \llbracket u_2\rrbracket)$
```

Algorithm 6: Wilcoxon signed-rank test

```
    Data: Paired value vectors $\llbracket\vec{a_1}\rrbracket$ and $\llbracket\vec{a_2}\rrbracket$ for $n$ subjects, mask vector $\llbracket\vec{m}\rrbracket$
    Result: Test statistic $\llbracket w\rrbracket$
1   $(\llbracket\vec{x_1}\rrbracket, \llbracket\vec{x_2}\rrbracket) \leftarrow \mathbf{cut}((\llbracket\vec{a_1}\rrbracket, \llbracket\vec{a_2}\rrbracket), \llbracket\vec{m}\rrbracket)$
2   $\llbracket\vec{d}\rrbracket \leftarrow \llbracket\vec{x_1}\rrbracket - \llbracket\vec{x_2}\rrbracket$
3   Let $\llbracket\vec{d'}\rrbracket$ be the absolute values and $\llbracket\vec{s}\rrbracket$ be the signs of elements of $\llbracket\vec{d}\rrbracket$
4   $\llbracket\vec{\hat{s}}\rrbracket \leftarrow \mathbf{sort}((\llbracket\vec{d'}\rrbracket, \llbracket\vec{s}\rrbracket))$
5   $\llbracket\vec{r}\rrbracket \leftarrow \mathbf{rank_0}(\llbracket\vec{\hat{s}}\rrbracket)$
6   return $\llbracket w\rrbracket \leftarrow \mathbf{sum}(\llbracket\vec{\hat{s}} \cdot \vec{r}\rrbracket)$
```

First, on line 3, both data vectors are cut based on the mask vector similarly to what was done in Algorithm 5. The signs are then sorted based on the absolute values $\llbracket\vec{d'}\rrbracket$ (line 4) and the ranking function $\mathbf{rank_0}$ is called. This ranking function differs from the function **rank** because we need to exclude the differences that have the value 0. Let the number of 0 values in vector $\llbracket\vec{d}\rrbracket$ be $\llbracket k\rrbracket$. As $\llbracket\vec{d}\rrbracket$ has been sorted based on absolute values, the 0 values are at the beginning of the vector so it is possible to use $\llbracket k\rrbracket$ as the offset for our ranks. Function $\mathbf{rank_0}$ assigns $\llbracket\vec{r}\rrbracket[i] \leftarrow 0$ while $\llbracket\vec{\hat{s}}[i] = 0\rrbracket$, and works similarly to **rank** on the rest of the vector $\llbracket\vec{\hat{s}}\rrbracket$, with the difference that $i \in \{1, \ldots, \llbracket n-k\rrbracket\}$.

2.2.2 The χ^2-Tests for Consistency.

If the attribute values are discrete such as income categories then it is impossible to apply t-tests or their non-parametric counterparts and we have to analyse frequencies of certain values in the dataset. The corresponding statistical test is known as χ^2-test.

The standard χ^2-test statistic is computed as

$$\chi^2 = \sum_{i=1}^{k}\sum_{j=1}^{2} \frac{(f_{ji} - e_{ji})^2}{e_{ji}} \quad ,$$

where f_{ji} is the observed frequency and e_{ji} is the expected frequency of the i-th option and j-th group. For simplification, we denote $c_i = f_{1i}$ and $d_i = f_{2i}$, then the frequencies can be presented as the contingency table 2.1. Let p_i be the sum of column i, r_j be the sum of row j and n be the number of all observations. The estimated frequency e_{ji} is computed as

$$e_{ji} = \frac{p_i \cdot r_j}{n} \quad .$$

	Option 1	**Option 2**	...	**Total**
Cases	c_1	c_2	...	r_1
Controls	d_1	d_2	...	r_2
Total	p_1	p_2	...	n

Table 2.1: Contingency table for the standard χ^2 test

```
Algorithm 7: $\chi^2$ test
  Data: Value vector $[\![\vec{a}]\!]$, corresponding mask vectors $[\![\vec{m_1}]\!]$ and $[\![\vec{m_2}]\!]$ for cases and controls
        respectively and a contingency table $[\![\mathbf{C}]\!]$ of size $2 \times k$
  Result: The test statistic $\chi^2$
1 Let $[\![n]\!]$ be the total count of elements
2 Let $[\![r_1]\!]$ and $[\![r_2]\!]$ be the row subtotals and $[\![p_1]\!], \dots, [\![p_k]\!]$ be the column subtotals
3 Let $[\![\mathbf{E}]\!]$ be a table of expected frequencies such that $[\![\mathbf{E}]\!][i][j] = \frac{[\![\vec{r_i}]\!] \cdot [\![\vec{p_j}]\!]}{n}$
4 $[\![\chi^2]\!] = \sum_{j=1}^{k} \frac{([\![\mathbf{C}]\!][1][j] - [\![\mathbf{E}]\!][1][j])^2}{[\![\mathbf{E}]\!][1][j]} + \frac{([\![\mathbf{C}]\!][2][j] - [\![\mathbf{E}]\!][2][j])^2}{[\![\mathbf{E}]\!][2][j]}$
5 return $[\![\chi^2]\!]$
```

Algorithm 7 shows how to calculate the χ^2 test statistic based on a contingency table. If the null hypothesis is supported if the computed χ^2 value does not exceed the critical value from the χ^2 table with $k-1$ degrees of freedom.

Often, the number of options in the contingency table is two—subjects who have a certain property and those who do not. Therefore, we look at an optimised version of this algorithm that works where the number of options in our test be 2. Then the test statistic can be simplified and written as

$$[\![t]\!] = [\![\frac{(c_1 + d_1 + c_2 + d_2)(d_1 c_2 - c_1 d_2)^2}{(c_1 + d_1)(c_1 + c_2)(d_1 + d_2)(c_2 + d_2)}]\!] \ .$$

The privacy-preserving version of the χ^2-test is implemented simply by evaluating the algorithm using SMC operations.

2.3 Conclusion

As a result, we were able to conduct a full-scale experimental statistical study so that confidential data were always processed using SMC. The strengths of our solution are generality, precision and practicality. We show that secure multi-party computation is flexible enough for implementing complex applications.

The work described in this chapter is discussed in more detail in the paper [10] that is available in Appendix **??** of this deliverable.

Chapter 3

Transformation-based Linear Programming

This chapter introduces some problems of the transformation-based linear programming that have been present in the previous works and demonstrates its insecurity. It presents concrete attacks against published methods following this approach. It has been proven that there are issues that cannot be resolved at all using the particular known class of efficient transformations that has been used before.

3.1 Privacy-preserving linear programming

We consider linear programming tasks in the canonical form

$$\text{minimize } \mathbf{c}^{\mathrm{T}} \cdot \mathbf{x}, \text{ subject to } A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0} \ . \tag{3.1}$$

Here A is an $m \times n$ matrix ($m \leq n$), $\mathbf{b}$ is a vector of length m and $\mathbf{c}$ is a vector of length n. There are n variables in the vector $\mathbf{x}$. Without lessening of generality we may assume that the quantity to be minimized is just the variable x_n, i.e. the vector $\mathbf{c}$ is of the form $(0, \ldots, 0, 1)^{\mathrm{T}}$. Any linear programming task can be brought to such a form by introducing a new variable w and adding the equation $\mathbf{c}^{\mathrm{T}} \cdot \mathbf{x} - w = w_0$ to the constraints, where $w_0 \in \mathbb{R}$ is determined (from out-of-band information about A, $\mathbf{b}$ and $\mathbf{c}$) so, that the minimal possible value of $\mathbf{c}^{\mathrm{T}} \cdot \mathbf{x} - w_0$ will certainly be positive.

The canonical form (3.1) of LP is equivalent to its *standard form*

$$\text{maximize } \mathbf{c}^{\mathrm{T}} \cdot \mathbf{x}, \text{ subject to } A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0} \ . \tag{3.2}$$

Indeed, the inequalities of the canonical form may be replaced with equalities by introducing slack variables. Each equality may be substituted with two inequalities of opposite directions.

3.2 Attacks against Transformation-based Linear Programming

In [32] we demonstrate a number of attacks against proposed protocols for privacy-preserving linear programming based on publishing and solving a transformed version of the problem instance. Our attacks exploit the geometric structure of the problem, which has mostly been overlooked in the previous analyses and is largely preserved by the proposed transformations.

3.2.1 Transformations Used in the Previous Works

Let a linear programming task be given in its standard form (3.2). The main basic transformations from the related works are the following.

Multiplying from the left. Multiplying A and $\mathbf{b}$ by a random invertible matrix P from the left does not change the feasible region of the linear program at all, and it remains revealed. All the solutions to the system, including the optimal solution, remain the same.

Multiplying from the right. Multiplying A and $\mathbf{c}$ by a positive monomial matrix Q from the right results in scaling and permuting the variables.

Shifting In some works, the initial variable vector $\mathbf{x}$ is not only scaled, but also shifted. This is done by introducing special slack variables for each shifted variable.

3.2.2 The Problems of Slack Variables

We introduce an attack that allows to remove the scaling and the permutation of variables. This is possible in the setting where all the constraints are represented by inequalities, one of the parties knows at least two inequality constraints, and the locations of the slack variables can be traced down. Our attack is polynomial-time. Let us state this problem in general.

Suppose that the initial linear program is given in its canonical form (3.1). In the standard approach proposed by the previous works, the inequalities are transformed to equalities by bringing the linear program to its standard form (3.2) introducing slack variables $\mathbf{x_s}$.

$$\text{maximize } \begin{pmatrix}\mathbf{c}\\ \mathbf{0}\end{pmatrix}^{\mathrm{T}} \cdot \begin{pmatrix}\mathbf{x}\\ \mathbf{x_s}\end{pmatrix}, \text{ subject to } \begin{pmatrix}A & I\end{pmatrix}\begin{pmatrix}\mathbf{x}\\ \mathbf{x_s}\end{pmatrix} = \mathbf{b}, \begin{pmatrix}\mathbf{x}\\ \mathbf{x_s}\end{pmatrix} \geq \mathbf{0}\ .$$

The columns of the matrix are first being scaled and permuted multiplying it by a monomial matrix Q from the right. Then it is multiplied by a random invertible matrix P from the left.

If the vector $\mathbf{c}$ is treated separately from the constraints, as it was done in the previous works, then the locations of the slack variables are clearly visible even after the permutation since their values in the cost vector are 0, and scaling does not affect 0 in any way. This issue allows to use row operations to reduce the constraints back to the standard form $\begin{pmatrix}A' & I\end{pmatrix}\begin{pmatrix}\mathbf{x}\\ \mathbf{x_s}\end{pmatrix} = \mathbf{b}'$. Here A' and $\mathbf{b}'$ may be different from A and $\mathbf{b}$. However, since we know that the feasible region is just scaled, the inequalities $A'\mathbf{x} \leq \mathbf{b}'$ define the scaled polyhedron of the initial program. Then we may use the knowledge about the initial inequalities to cancel out the scaling and permutation. This attack is described more precisely in [32].

The locations of slack variables can be potentially hidden by adding the equation $\mathbf{c}^{\mathrm{T}} \cdot \mathbf{x} - w = w_0$ to the constraints and minimizing over the variable w only. However, we present attacks that detect the slack variables due to their special behaviour.

1. If the entries of P and Q are sampled from a uniform or a (folded) normal distribution, the columns that correspond to the slack variables have in general greater variance. This problem can be resolved by bringing the matrix A to its reduced row echelon form instead of multiplying it by P. However, since this form has to be computed in a privacy-preserving way, this transformation is more expensive.

2. The other attack is based on the geometric properties of the feasible region. It turns out that, optimizing the LP task in random directions, slack variables tend to take the value 0 much more often than the initial variables. The attack is efficient in practice, and for numerous LP tasks the slack variables have been located perfectly all at once. The particular results can be seen in [32].

The slack variables also allow to undo shifting. Namely, although the values of the initial variables $\mathbf{x}$ are hidden by a randomness vector $\mathbf{r}$, the scaled values of $\mathbf{x}$ are leaked into the special slack variables $\mathbf{s}$ that have been introduced by shifting. Distinguishing slack and non-slack variables in this case is especially easy due to the special relations between these variables. After removing non-slack variables by gaussian elimination, we obtain an LP with variables $\mathbf{s}$ whose feasible region is just a scaled version of the original one, without shifting. The attack is described in more details in [32].

3.3 Impossibility of Secure Transformation-based Linear Programming

As a conclusion of the previous section, the attacks are efficient in practice and cast serious doubt to the viability of transformation-based approaches in general. In our next paper [33], we study the security definitions and methods for transformation-based outsourcing of linear programming in general. The recent attacks have shown the deficiencies of existing security definitions; thus we propose a stronger, indistinguishability-based definition of security of problem transformations that is very similar to IND-CPA security of encryption systems. We study the realizability of this definition for linear programming and find that barring radically new ideas, there cannot exist transformations that are secure information-theoretically or even computationally.

The privacy requirements for our task are the following. The sizes m and n, and the bounding box of the feasible region are public. Matrix A and the vector $\mathbf{b}$ have to remain secret. The solution $\mathbf{x}_{\text{opt}}$ may become public. We wish to transform the linear programming task at hand to a task "minimize $\mathbf{c}'^{\mathrm{T}} \cdot \mathbf{y}$, subject to $A'\mathbf{y} = \mathbf{b}'$, $\mathbf{y} \geq \mathbf{0}$", such that from the solution $\mathbf{y}_{\text{opt}}$ and secret data generated during the transformation, we could efficiently recover the solution $\mathbf{x}_{\text{opt}}$ to the original task.

Quite clearly, the transformations described in the previous subsection do not satisfy this privacy requirement. Since these transformations are all some instances of affine transformations (linear transformation + shifting), we have studied if a more general class of affine transformations can satisfy our security requirement.

- **Information-theoretic Security** First of all, we studied if information-theoretic security is possible. Since computational security would most probably require to introduce some new assumptions (since cryptography over real numbers has not received too much attention so far), achieving information-theoretic or at least statistical security would be really preferable. Theoretically, it could be achieved by introducing more dimensions to the feasible region and encoding a scaled/shifted initial feasible region as some projection. However, even if we are able to encode all possible shapes as some non-trivial projection, we will still have problems with scaling. The idea behind the proof is that an affine transformation preserves distances between the hyperplanes, and although the distances in the transformed feasible region may also depend on the randomness, they still depend also on the initial distances, what makes the difference traceable in average. The particular construction used as a counterexample can be seen in [33].

  Since in our counterexample we had to use the assumption of perfect secrecy, we did not give up and started studying computational security.

- **Computational Security** We could still hope to hide the initial feasible region by using an affine transformation in such a way that it would be computationally difficult to recover it. We studied the methods used in the previous works and proposed our own affine transformations, but they still were obviously vulnerable. We tried to understand what is the reason why all the proposed methods fail. The problem is that all the hiding except scaling and permutation requires introducing additional variables (even multiplication by an invertible matrix from the left in general cannot be done without introducing slack variables first). And both in the previous works and in our solutions, these new variables have been distinguishable by their behaviour, so that permutation has not helped to hide them. After being revealed, these variables can be removed by gaussian elimination, thus undoing all the hiding that they provide.

  For that reason, we empirically state the necessary requirement: for a security parameter t, any set of t variables should look the same to the adversary. However, if we want this claim to hold for a reasonable t, it should hold at least for $t = 2$. We have tried to define the properties of a feasible region that would satisfy this requirement. It has turned out that the only suitable geometric shape is a simplex: a polyhedron defined by an equation $\{(x_1, \ldots, x_n) \mid x_1 + \ldots + x_n \leq c\}$ and inequalities $x_1 > 0, \ldots, x_n \geq 0$ for some $c \geq 0$. Such a linear program has only one dimension and hence cannot be used to encode anything reasonable. The full proof can be found in [33].

3.4 Conclusions

We have shown that the current approaches towards transformation-based privacy-preserving outsourcing or multiparty linear programming are unlikely to be successful. Success in this direction requires some radically new ideas in transforming polyhedra and/or in cryptographic foundations violating the rather generous assumptions we have made. We conclude that for solving linear programming problems in privacy-preserving manner, cryptographic methods for securely implementing Simplex or some other linear programming solving algorithm are the only viable approach.

Chapter 4

Privacy-Preserving Execution of Finite Automata

A *deterministic finite automaton (DFA)* over an alphabet Σ is a tuple $A = (Q, q_0, \delta, F)$, where Q is the set of *states*, $q_0 \in Q$ is the *initial state* of the automaton, $F \subseteq Q$ is the set of *accepting states* and $\delta : Q \times \Sigma \to Q$ is the *transition function*. The transition function can be extended to have the type $Q \times \Sigma^* \to Q$, by defining $\delta(q, \varepsilon) = q$ and $\delta(q, sa) = \delta(\delta(q, s), a)$ for all $q \in Q$, $s \in \Sigma^*$ and $a \in \Sigma$. The automaton *accepts* a string s if $\delta(q_0, s) \in F$.

A *non-deterministic finite automaton (NFA)* is also a tuple $A = (Q, q_0, \delta, F)$, but now δ is a function with the type $Q \times \Sigma \to 2^Q$. It can again be extended to the arguments of type $Q \times \Sigma^*$ by defining $\delta(q, \varepsilon) = \{q\}$ and $\delta(q, sa) = \bigcup_{q' \in \delta(q,s)} \delta(q', a)$. The automaton *accepts* a string s if $\delta(q_0, s) \cap F \neq \emptyset$.

4.1 Problem description

We have a private string over the alphabet Σ (which is public). As it is difficult to hide the size of inputs without applying a lot of padding, we assume that the length of the string is public. The individual characters, however, are sensitive data.

We also have a private finite automaton. Again, we are not trying to hide the size of our inputs, hence we assume that the number of states $|Q|$ is public. The transfer function δ and the set of accepting states F are, however, private.

We want to compute whether the automaton accepts the string. The result must remain private, too.

"Privacy" may mean many different things, depending on the number of parties in the system, the coalitions which know or may know different data items, and the potential collusions between parties. We are considering the most general setup described also in D5.2.1 [7, Chap. 1]. We are working in the *Arithmetic Black Box (ABB)* model. In this model, the protocol set for SMC, run by the parties, is such that some kind of *private storage* is implemented. Values held in this private storage can become public only if several of the computing parties cooperate. The input string and the description of the automaton are kept in this private storage. We want to execute the automaton in this string so, that the result of this execution is added to the private storage while nothing about the inputs is made public. Private computation tasks implemented in such manner are universally composable, meaning that the security of some composition of these tasks follows from the security of each task separately.

4.2 Private selection

If privacy were not an issue, then one checks whether a DFA A accepts a string $s = a_1 \cdots a_\ell$ by computing $q_1 = \delta(q_0, a_1), q_2 = \delta(q_1, a_2), \ldots, q_\ell = \delta(q_{\ell-1}, a_\ell)$ and checks whether $q_\ell \in F$. The function δ is typically given in *tabular form*. I.e. to find $\delta(q, a)$, one has to locate the cell (q, a) of δ and read its contents. Such operation is typically much more expensive if the index of the cell is private. Indeed, in this case, the selection algorithm must "touch" all[1] cells of δ, otherwise one would reveal which cells definitely do not correspond to (q, a). At each cell, the selection algorithm typically has to perform some *non-free* operations with private values[2]. A typical example of private selection performs the scalar product of the table with the characteristic vector of the index. The computation of the characteristic vector requires $\Omega(|\delta|)$ work.

We have shown how almost all expensive operations of a private selection can be moved *offline*, i.e. performed before the automaton and/or the input string are available [35]. In the online phase, we have to perform only a couple of multiplications of private values, irrespective of the sizes of Q and Σ. We will now describe our protocols for that.

We use the usual notation $[\![v]\!]$ for the value v stored in the ABB. The notation $[\![v_1]\!]$ *op* $[\![v_2]\!]$ denotes the computation of v_1 *op* v_2 by the ABB (translated to a protocol in the implementation of the ABB). Let a private table v with m elements be given, let the indices of the cells be $i_1, \ldots, i_m$ (these indices are public and we require them to be non-zero). We require that both $i_1, \ldots, i_m$ and the elements of the table $v_{i_1}, \ldots, v_{i_m}$ are elements of a finite field $\mathbb{F}$ with at least $m+1$ elements. There exist protocols for generating a uniformly random element of $\mathbb{F}$ inside the ABB (denote: $[\![r]\!] \overset{\$}{\leftarrow} \mathbb{F}$), and for generating a uniformly random non-zero element of $\mathbb{F}$ together with its inverse (denote: $([\![r]\!], [\![r^{-1}]\!]) \overset{\$}{\leftarrow} \mathbb{F}^*$). These protocols require a small constant number of multiplications on average for any ABB [16]. For any $\mathbf{I} = \{i_1, \ldots, i_m\}$ there also exist Lagrange interpolation coefficients $\lambda^{\mathbf{I}}_{j,k}$ depending only on the set $\mathbf{I}$, such that for any polynomial V over $\mathbb{F}$ with degree at most $m-1$ we have $V(x) = \sum_{j=0}^{m-1} c_j x^j$, where $c_j = \sum_{k=1}^{m} \lambda^{\mathbf{I}}_{j,k} V(i_k)$. These coefficients are public and can be computed in the offline phase, too.

Our private selection algorithm Alg. 8 receives as inputs the private values $[\![v_{i_1}]\!], \ldots, [\![v_{i_m}]\!]$ and the private index $[\![j]\!]$, where $j \in \{i_1, \ldots, i_m\}$. It responds with the private value $[\![v_j]\!]$. The work of this algorithm is divided into three phases. During the vector-only phase the table $([\![v_{i_1}]\!], \ldots, [\![v_{i_m}]\!])$ is available, while $[\![j]\!]$ can be used only in the online phase. This corresponds to common use cases (e.g. spam filtering), where the DFA is known before the actual input string.

Algorithm 8: Private selection protocol

```
   Data: Vector of indices $i_1, \ldots, i_m \in \mathbb{F}\backslash\{0\}$
   Data: Vector of values $([\![v_{i_1}]\!], \ldots, [\![v_{i_m}]\!])$ with $v_{i_1}, \ldots, v_{i_m} \in \mathbb{F}$.
   Data: Index $[\![j]\!]$ to be looked up, with $j \in \{i_1, \ldots, i_m\}$.
   Result: The looked up value $[\![w]\!] = [\![v_j]\!]$.
1  Offline phase
2  $([\![r]\!], [\![r^{-1}]\!]) \overset{\$}{\leftarrow} \mathbb{F}^*$
3  for $k = 2$ to $m-1$ do $[\![r^j]\!] \leftarrow [\![r]\!] \cdot [\![r^{j-1}]\!]$;
4  Compute the coefficients $\lambda^{\mathbf{I}}_{j,k}$ from $i_1, \ldots, i_m$.
5  Vector-only phase
6  foreach $k \in \{0, \ldots, m-1\}$ do $[\![c_k]\!] \leftarrow \sum_{l=1}^{m} \lambda^{\mathbf{I}}_{k,l} [\![v_l]\!]$;
7  foreach $k \in \{0, \ldots, m-1\}$ do $[\![y_k]\!] \leftarrow [\![c_k]\!] \cdot [\![r^k]\!]$;
8  Online phase
9  $z \leftarrow$ retrieve$([\![j]\!] \cdot [\![r^{-1}]\!])$
10 $[\![w]\!] = \sum_{k=0}^{m-1} z^k [\![y_k]\!]$
```

Algorithm Alg. 8 represents the vector $(v_{i_1}, \ldots, v_{i_m})$ as a polynomial $V(x) = \sum_{k=0}^{m-1} c_k x_k$, such that $V(i_k) = v_{i_k}$. The value $V(j)$ is computed as $\sum_{k=0}^{m-1} (jr^{-1})^k (c_k r^k)$, where r is a random non-zero element of $\mathbb{F}$. In this way, jr^{-1} is also a random non-zero element of $\mathbb{F}$ and may be made public. The private random element r together with its inverse r^{-1} and its powers can be computed in the offline phase while the coefficients c_i of V and the products $c_k r^k$ can be computed in the vector-only phase. In the online

[1]Unless the table has been somehow scrambled before, as in implementations of *Oblivious RAM* [20]

[2]All operations except the computation of linear combinations of private values with public coefficients; which does not require any expensive computations or communication in any existing implementations of ABBs

phase, we perform a single multiplication (to find jr^{-1}) with private values, and a single declassification. Algorithm Alg. 8 can be used with any ABB implementation that operates on elements of a finite field of sufficient size, and it inherits the security guarantees of the ABB.

The complexity of private selection is thus shifted to the offline and vector-only phases. In the vector-only phase we still perform m multiplications with private values (while computing $[\![y_k]\!]$). For certain ABB implementations, it is possible to avoid that cost. Namely, for ABBs based on Shamir's secret sharing [45] and using the Gennaro-Rabin-Rabin multiplication protocol [26], the computation of the scalar product of two private vectors is no more expensive than a single multiplication of private values. We hence rewrite the vector-only and online phases of the private selection protocol as depicted in Alg. 9.

Algorithm 9: Improved vector-only and online phases of the private lookup protocol

```
  Data: Lagrange interpolation coefficients $\lambda^{\mathbf{I}}_{j,k}$
  Data: Random non-zero $[\![r]\!]$ and its powers $[\![r^{-1}]\!], [\![r^2]\!], \ldots, [\![r^{m-1}]\!]$.
  Data: Vector of values $([\![v_{i_1}]\!], \ldots, [\![v_{i_m}]\!])$ with $v_{i_1}, \ldots, v_{i_m} \in \mathbb{F}$.
  Data: Index $[\![j]\!]$ to be looked up, with $j \in \{i_1, \ldots, i_m\}$.
  Result: The looked up value $[\![w]\!] = [\![v_j]\!]$.
1 Vector-only phase
2 foreach $k \in \{0, \ldots, m-1\}$ do $[\![c_k]\!] \leftarrow \sum_{l=1}^{m} \lambda^{\mathbf{I}}_{k,l} [\![v_l]\!]$;
3 Online phase
4 $z \leftarrow \mathsf{retrieve}([\![j]\!] \cdot [\![r^{-1}]\!])$
5 foreach $j \in \{0, \ldots, m-1\}$ do $[\![\zeta_j]\!] \leftarrow z^j [\![r^j]\!]$;
6 $[\![w]\!] = ([\![c_0]\!], \ldots, [\![c_{m-1}]\!]) \cdot ([\![\zeta_0]\!], \ldots, [\![\zeta_{m-1}]\!])$
```

Compared to Alg. 8, we have moved the entire computation of the products $z^j [\![c_j]\!][\![r^j]\!]$ to the online phase, thereby reducing the vector-only phase to the computation of certain linear combinations. The complexity of the online phase has increased by the computation of the scalar product in the last line. The total cost of the online phase is thus two multiplications and one declassification.

4.3 DFA execution

Private selection is all that we need to execute a private DFA on a private string. For a string of length ℓ, we sequentially perform ℓ private selections to find $[\![q_1]\!], \ldots, [\![q_\ell]\!]$. Finally, we will perform one more private selection on the characteristic vector of the set of accepting states F, using q_ℓ as the index. In this way, we have emulated the sequential DFA execution algorithm.

One can also execute a DFA $A = (Q, q_0, \delta, F)$ on a string $s = a_1, \ldots, a_\ell$ in a parallel manner. For any $t \in \Sigma^*$, let $\delta(\cdot, t) : Q \to Q$ be the mapping we get from δ (extended to $Q \times \Sigma^*$) by fixing its second argument as t. The function $\delta(\cdot, \varepsilon)$ is the identity function on Q and the equality $\delta(\cdot, t_1 t_2) = \delta(\cdot, t_2) \circ \delta(\cdot, t_1)$ holds for all $t_1, t_2 \in \Sigma^*$. If all functions are represented in tabular form, then the computation of the composition requires $|Q|$ lookups. We can now compute $\delta(\cdot, s)$ in a divide-and-conquer fashion, and then check whether $\delta(q_0, s) \in F$. The computation requires $\log \ell$ "steps". The total amount of work is $|Q|$ times the work performed by the sequential execution.

We have not implemented the parallel version of the DFA execution algorithm in privacy-preserving manner, using our private selection algorithm as a subroutine. For certain parameters (small —Q— and large ℓ), this algorithm may in practice perform much better than the sequential algorithm, due to its smaller round complexity.

4.4 NFA execution

Due to their non-deterministic nature, NFAs are more complicated to handle in a secure manner. We see that even though the NFA execution starts from a single state, after the intermediate steps it can generally be in a subset of states. In order to account for this, we will use characteristic vectors of the intermediate sets $\mathcal{Q}_i = \boldsymbol{\delta}_A(a_1 \cdots a_i)$ to represent them. Let $\mathbf{q}^i = (q^i_0, q^i_1, \ldots, q^i_{m-1})$ be a binary vector, where $q^i_j = 1$ iff the state $q_j \in \mathcal{Q}_i$.

As $\mathcal{Q}_0 = \{q_0\}$, we have $\mathbf{q}^0 = (1, 0, \ldots, 0)$. Subsequent $\mathbf{q}^i$-s will depend both on the given automaton A and the string s. Namely, in order to determine $\mathbf{q}^i$ from $\mathbf{q}^{i-1}$, δ and a_i, we can compute

$$q^i_j = \bigvee_{q \in \mathcal{Q}_{i-1}} [q_j \in \delta(q, a_i)] = \bigvee_{k=0}^{m-1} q^{i-1}_k \& [q_j \in \delta(q_k, a_i)] \tag{4.1}$$

for all the components q^i_j of the characteristic vector $\mathbf{q}^i$.

In order to determine efficiently whether $q_j \in \delta(q_k, a_i)$, we need an efficient representation of δ as well. We will represent it as a look-up table $\overline{\delta} : Q \times Q \to \mathcal{P}(\Sigma)$, where $a_i \in \overline{\delta}(q_k, q_j)$ iff $q_j \in \delta(q_k, a_i)$. To encode subsets of Σ, we will once again use characteristic vectors; let $\mathcal{S} \subseteq \Sigma$ be encoded by vector $\mathbf{s} = (s_1, \ldots, s_n)$ where $s_i = 1$ iff the corresponding $\sigma_i \in \mathcal{S}$. Similarly, we also represent the characters of the string using binary characteristic vectors $\mathbf{a}_1, \ldots, \mathbf{a}_\ell$, where $\mathbf{a}_i = (a^1_i, \ldots, a^n_i)$ and $a^j_i = 1$ iff $a_i = \sigma_j$. As a result, the value of the predicate $q_j \in \delta(q_k, a_i)$ can be computed as a dot product $\overline{\delta}(q_k, q_j) \cdot \mathbf{a}_i$.

The complexity of NFA execution is significantly higher than DFA execution. Some of its steps can also be performed offline. Namely, the disjunction in (4.1) can be computed by adding up the elements and comparing the result to 0 (and flipping the outcome) [36]. Working in a suitable field, comparison to 0 may be implemented using just one round of online multiplications using the protocol by Lipmaa and Toft [39] (though some precomputation is necessary).

4.5 Applications of our results

We have shown that arithmetic black boxes support fast lookups from private tables according to a private index. We have used this operation to obtain very efficient DFA execution algorithms. Our results show that for private lookups in an ABB, complex techniques based on Oblivious RAMs [20] are not necessary. We expect our techniques to have wide applicability in privately processing graph-like data structures.

Chapter 5

Public Verifiability for Parties in SMC

In this chapter we propose a method that allows to detect the parties that have violated the protocol rules after the computation has ended, thus making the protocol secure against covert attacks. This approach can be useful in the settings where for any party it is fatal to be accused in violating protocol rules. In this way, up to the verification, all the computation can be performed in semi-honest model, which makes it very efficient in practice. The verification is statistical zero-knowledge, and it it based on linear probabilistically checkable proofs (PCP) for verifiable computation. Hence each malicious party is detected with probability $1 - \varepsilon$ for a negligible ε that is defined by the failure of the corresponding linear PCP. The initial protocol has to be executed only once, and the verification requires in total 3 additional rounds. The verification also ensures that all the parties have sampled all the randomness from an appropriate distribution. Its efficiency does not depend on whether the inputs of the parties have been shared, or each party uses its own private input.

5.1 Introduction

The semi-honest and the malicious model are the two main models in which cryptographic protocols are studied. In the semi-honest model, the adversary is curious about the values it gets, and it tries to extract information out of them, but it follows the protocol rules honestly. In the malicious model, the adversary is allowed to do whatever it wants. In addition to these traditional models, a notion of covert security was proposed in [2]. In this model, the adversary is malicious, but it will not cheat if it will be caught with a non-negligible probability, which can be defined more precisely as a security parameter. This notion is very realistic in many computational models, where the participants care about their reputation and will not cheat even if this probability is not close to 1.

Some works have been dedicated to covert security [37, 18], where [37] treats the security for two-party computation based on garbled circuits, both the covert and the malicious cases, and [18] deals with honest majority protocols for an arbitrary number of parties. A more precise definition of *covert security with public verifiability* has been proposed in [1]. This allows the cheater to be blamed publicly.

5.2 Our Contribution

In this work we propose a scheme that is based on succinct computation verification. Our work is closely related to [18] that is dealing with honest majority protocols for an arbitrary number of parties. The solution proposed in [18] is based on running the initial protocol on two inputs, the real shares and the dummy shares. In this case, the real shares should be indistinguishable from random, and hence in the beginning the protocol is being rewritten to a shared form. Differently from [18], our solution does not require rewriting the original protocol. The original protocol has to be run only once, and each malicious party is detected with probability $1 - \varepsilon$ for a negligible ε. Our approach is statistical zero-knowledge, and it it based on linear probabilistically checkable proofs for verifiable computation. The particular PCP that we are using is the one proposed in [4]. The quantity ε is defined by the failure of the corresponding linear PCP behind the protocol. The verification requires in total 3 additional rounds. Additionally, it ensures that all the parties have sampled all the randomness from an appropriate distribution.

The major drawback of our scheme is that the number of values sent per one round is exponential in the number of parties. In [18], efficiency is achieved by reducing the probability of being detected from 1/2 to 1/4. We cannot use the same approach in our case since the probability of being detected would immediately become negligible. Nevertheless, the settings make the verification very efficient for a small number of parties.

Similarly to [18], we prove the security of our scheme in UC model [13].

5.3 Protocol Description

This section gives a general overview of the protocol. We state the assumptions on which our protocol is based, describe which precomputation has to be performed before running the original protocol, and which messages should be sent in addition to the initial ones during the execution. We briefly explain what happens in the final verification, without going into details. Similarly to [18], all the inputs and the communication values are committed, but in a special shared way, using signatures. Due to the sharing, the signatures do not have to hide the messages at all, and should be rather perfectly binding. The entire verification is still zero-knowledge.

5.3.1 Notation

Throughout this chapter, we use the following notation:

- the upper case letters A denote matrices;
- the bold lower case letters $\mathbf{b}$ denote vectors;
- $\langle \mathbf{a}, \mathbf{b} \rangle$ denotes the scalar product of $\mathbf{a}$ and $\mathbf{b}$;
- $(\mathbf{a}||\mathbf{b})$ is a concatenation of vectors $\mathbf{a}$ and $\mathbf{b}$.

5.3.2 Assumptions

Our verification protocol is based on security of some other schemes. Here is the list of used assumptions.

- Secure point-to-point channels between each pair of parties.
- Broadcast channels between subsets of parties.
- Honest Verifier Statistical Zero-Knowledge Linear Probabilistically Checkable Proofs for verifiable computation [38, 25, 4, 6]. In particular, all the complexity estimations in this chapter are based on the solution proposed in [4].
- Functionality that allows to prove to third parties which messages one received during the protocol, and to further transfer such revealed messages. This allows to protect the initial protocol from halting problem (when the computation cannot proceed due to some malicious party that is either just doing nothing, or causes some other party to wait by sending wrong messages). We use the solution proposed in [18].

5.3.3 The Protocol Outline

We describe briefly the initial settings, and how the new verifiable protocol differs from the original one.

- In the initial settings, we have a set of arithmetic circuits C_i^j over some finite field $\mathbb{F}$, where C_i^j is the circuit computed by the party M_i on the j-th round of computation. Some outputs of C_i^j may be used as inputs for some C_k^{j+1}, so there is some communication between the parties. Each circuit may use some randomness that comes from random uniform distribution in $\mathbb{F}$ (this is sufficient to model any other distribution). The circuits could be boolean as well, since there also exist linear probabilistically checkable proofs based on boolean circuits [38], so our verification is not restricted to computation over some certain field.

- The computation is performed by n parties. Let them be denoted M_i for $i \in \{1, \ldots, n\}$. A necessary condition is that at least $t = \lfloor n/2 \rfloor + 1$ are honest.

- Before the execution of original protocol starts, the inputs of the parties are committed in a special way. Let the input of the party M_i be represented by a vector $\mathbf{x}_i$ over $\mathbb{F}$. M_i represents $\mathbf{x}_i$ as $\binom{n-1}{t}$ distinct sums of the form $\mathbf{x}_i = \sum_{k \in T_j} \mathbf{x}_{ikT_j}$ for $j \in \left\{1, \ldots, \binom{n-1}{t}\right\}$, where each T_j represents a distinct subset of t other parties, and k corresponds to one particular party in that subset. The idea is that each party has to prove its honestness to any subset of t other parties. All the shares are signed and distributed amongst the corresponding parties. Although the number $\binom{n-1}{t}$ is exponential, computing all $t \cdot \binom{n-1}{t}$ signatures is not less efficient than computing just one, for example using hash Merkle tree.

- The randomness used in the protocols should also be committed in the same way. Moreover, we want to ensure that it indeed comes from random uniform distribution, without revealing to anyone its value.

  - Let an arbitrary set of t parties be responsible for generating the randomness. Let these parties be called "generators". By honest majority assumption, at least one of them is honest. For each M_i, they generate the randomness $\mathbf{r}_i$ as follows. Each generator M_j generates $\mathbf{r}_{ji}$ of the same length that $\mathbf{r}_i$ should be. The idea is to take $\mathbf{r}_i = \mathbf{r}_{i1} + \ldots + \mathbf{r}_{it}$. Since at least one party is honest, the vector $\mathbf{r}_i$ comes from a random uniform distribution.
  - Each generator M_j represents its $\mathbf{r}_{ij}$ as $\binom{n-1}{t}$ distinct sums of the form $\mathbf{r}_{ij} = \sum_{k \in T_\ell} \mathbf{r}_{ijkT_\ell}$ for $\ell \in \left\{1, \ldots, \binom{n-1}{t}\right\}$. All the shares are signed and sent to M_i. After M_i receives $\mathbf{r}_{ij}$ from all generators M_j, it may compute the sum of all $\mathbf{r}_{ij}$ and use it as $\mathbf{r}_i$ (M_i has to verify if the shares for different sets T_ℓ indeed all represent the same value). Then M_i signs all the received shares also by itself, and distributes the shares and the signatures (both signed by M_i and the corresponding generator M_j) amongst appropriate subsets of t parties, similarly to $\mathbf{x}_i$.

- The original protocol is computed in the same way as before. Additionally, each communicated vector $\mathbf{c}_{ij}^\ell$ sent by M_i to M_j on the round ℓ is presented as $\binom{n}{t}$ distinct sums $\mathbf{c}_{ij}^\ell = \sum_{k \in T_{j'}} \mathbf{c}_{ijkT_{j'}}^\ell$ (here we have $\binom{n}{t}$ instead of $\binom{n-1}{t}$ since both communicating parties should later verify the consistency of this value from each other). Along with each $\mathbf{c}_{ij}^\ell$, M_j receives the signature of $\mathbf{c}_{ij}^\ell$ and the signatures of all the shares $\mathbf{c}_{ij\ell kT_{j'}}^\ell$. M_j checks if the signatures are all indeed valid, and in turn signs them. M_j distributes the corresponding signatures (both signed by M_i and M_j) amongst each T_j. Here M_j is unable to check whether the shares under the signatures are valid and indeed sum up to $\mathbf{c}_{ij}^\ell$. All the shares will be distributed after the protocol execution, and then M_i may present the signature of $\mathbf{c}_{ij}^\ell$ to complain.

- After the protocol computation ends, all the communication shares are finally distributed. Each party M_j is verified for honestness. A party is honest iff it can prove that it acted according to the protocol, given the signed input, randomness, and communication that it had with the other parties. It has to perform a 3-round interactive proof with each subset of t parties in parallel. Since each subset of t parties holds all the shares of all the committed values, they are able to reconstruct the committed values and check if the proof indeed corresponds to them.

In general, in a linear PCP the prover has to prove the knowledge of a vector $\pi = (\mathbf{p}||\mathbf{d})$ such that certain combinations of $\langle \pi, \mathbf{q}_i \rangle$ for special challenges $\mathbf{q}_1, \dots, \mathbf{q}_5$ should be equal to 0, and $\mathbf{d}$ corresponds to the committed values. The problem is that the prover cannot see any of the $\mathbf{q}_i$ before committing the proof, but at the same time π should remain private.

In particular, for any subset of t verifiers, the following has to be done (ordered by rounds).

1. The verifiers agree on a random $\tau \in \mathbb{F}$ that is sufficient to generate all $\mathbf{q}_1, \dots, \mathbf{q}_5$ (in one round). The prover generates shares $\pi = \pi_1 + \ldots + \pi_t$ (where the $\mathbf{d}$ part is shared in the same way as it was committed to the given set of t verifiers) and distributes them amongst the parties. Each verifier checks if the part that corresponds to $\mathbf{d}$ is consistent with the signatures of shares sent during the computation.
2. Each verifier V_i computes and publishes $\langle \pi_i, \mathbf{q}_j \rangle$ for $j \in \{1, \dots, 5\}$. The τ is published. Everyone may compute $\langle \pi_1, \mathbf{q}_j \rangle + \ldots + \langle \pi_t, \mathbf{q}_j \rangle = \langle \pi, \mathbf{q}_j \rangle$ for $j \in \{1, \dots, 5\}$ and locally verify the necessary combinations. The prover checks if all the scalar products are computed correctly, and complains if necessary.

A party is claimed honest iff it succeeds in all the $\binom{n-1}{t}$ proofs against t other parties. This means that even if it was in collaboration with $t-2$ other malicious parties, there exists a subset of t all-honest parties that will definitely accept only the correct proof. We also need to ensure that the presence of malicious parties will not make the proof fail for an honest prover, and this can be done by revealing the signatures that correspond to the shares of incorrect scalar products. An honest party is safe to open them since they are known by the adversary anyway. The details of accusations and the security proofs can be seen in [34].

5.3.4 Properties

In our settings, we have n parties M_i. Compared to the original protocol, for each M_i the proposed solution has the following computational overheads.

- Let $p = \binom{n-2}{t-1}$. This is the number of t-sets in which one party participates as a verifier. If everyone is honest, then in order to verify M_j's honestness, M_i has to send the following messages:
  - In the initial protocol, send two signatures and $tp+p$ vectors of length $O(|C|)$ to each of the $n-1$ parties (tp for the randomness, and p for the inputs).
  - During the protocol execution, in addition to the original protocol communication, send $r(1+n)$ signatures to each of the $n-1$ receiver parties, where r is the number of rounds (one signature for the entire message and n for its shares). Each receiver M_j produces rn more signatures of the same values (that correspond to the shares). All these signatures are distributed by each receiver M_j amongst corresponding $n-1$ remaining parties (including M_i).
  - After the protocol execution, compute locally the auxiliary values for the proof in $O(|C| \log |C|)$ steps, as shown in [4]. Send to each of the other $n-1$ parties in parallel $1 + (n-1)$ signatures and $p + p(n-1)$ vectors of length $O(|C|)$: p for intermediate variables (for each proof separately, signed with one signature), and $p(n-1)$ for communication.

  As a verifier, in the verification process each M_i has to do the following:

  - Locally generate, sign and broadcast a random element of $\mathbb{F}$.
  - Locally generate p state informations $\mathbf{u}$ and $5p$ challenge vectors $\mathbf{q}_k$ of length $O(|C|)$ (according to the arithmetic circuit). This can be done in $O(|C|)$ steps, as shown in [4].

  - Locally sum up p times t^2 vectors of length $O(|C|)$.
  - Locally concatenate 4 vectors of total result length $O(|C|)$: the shares of the input, randomness, communication, and the intermediate values. This is done p times, for each verification set.
  - Locally compute $5p$ scalar products of vectors of length $O(|C|)$ and broadcast them (5 for each proof).
  - In the end, compute a constant number of local operations based on these scalar products: 2 multiplications, 3 additions, 1 scalar product of length $O(|\mathbf{v}|)$ for the part of the input $\mathbf{v}$ whose value is public (which is in general just the constant 1), all operations in $\mathbb{F}$. Everything is done p times, for each proof.

- If something goes wrong with the proof of M_j's honestness, then in the worst case each sent message has to be sent in such a way that it is possible to prove afterwards what has been sent to whom. The $\mathcal{F}_{transmit}$ functionality from [18] requires each message to be broadcast to all $n-1$ parties, and then this message should be delivered by each of the $n-2$ remaining parties to the receiver. No additional signatures are needed since we have already considered all of them in the case where everyone acts honestly.

According to the Linear PCP description from [4], a dishonest prover may cheat with probability $\frac{2m}{|\mathbb{F}|}$ where m is the number of multiplication gates in the circuit. This means that either the field should be large enough, or the verification should be repeated k times, so that $\left(\frac{2m}{|\mathbb{F}|}\right)^k$ is negligible. All the k verifications can be done in parallel, by generating k sets of challenges instead of one, thus do not increasing the number of rounds at all, and increasing the communication in total by $p(n-1)k$ field elements and $p(n-1)k$ proof vector shares.

5.4 Using the Proposed Protocol in Secure Multiparty Computation Platforms

In this section we discuss how the proposed verification could be used in Secure Multiparty Computation Platforms. More precisely, here we should consider the case where in addition to *computing* parties (that participate in the protocol) we may have *input* parties (that provide the inputs, sharing them in some way amongst the computing parties) and the *result* parties (that receive the final output). In our protocol, the computing parties do commit the inputs before the computation starts, but we must ensure that these are indeed the same inputs that have been provided by the input parties.

5.4.1 Treating Inputs/Outputs as Communication

As a simpler solution, we may just handle the input and the output similarly to communication. Hence the following enhancements are made.

1. Let the number of input parties be N. In the beginning, each input party P_i generates the shares $\mathbf{x}_{i1}, \ldots, \mathbf{x}_{in}$ (according to an arbitrary sharing scheme) from all the computing parties $M_1, \ldots, M_n$, as it would do without the verification. Each M_j should now use the input vector $\mathbf{x}_j = (\mathbf{x}_{1j}|| \ldots ||\mathbf{x}_{Nj})$, where each $\mathbf{x}_{ij}$ is provided by an input party P_i. Now, for each $\mathbf{x}_{ij}$, P_i generates by itself all the $\binom{n-1}{t}$ shares $\mathbf{x}_{ijkT_\ell}$ such that $\sum_{k \in T_j} \mathbf{x}_{ijkT_\ell} = \mathbf{x}_{ij}$, signs all these shares, and sends them to M_j. As before, M_j should also sign all of these shares before redistributing them amongst all the verifier t-sets. The verifiers should now check both signatures, similarly to how it was done to communication.

2. In the end, each receiver party R_i gets the shares $\mathbf{y}_{i1}, \ldots, \mathbf{y}_{in}$ from all the computing parties $M_1, \ldots, M_n$. Now each M_j has to generate $\binom{n-1}{t}$ sums $\sum_{k \in T_j} \mathbf{y}_{ijkT_\ell} = \mathbf{y}_{ij}$, exactly in the same way as it would do with an ordinary communication value. M_j sends the shares and their signatures to P_i, and P_i redistributes them amongst the t-sets. In the verification process, they check both signatures, similarly to how it was done to communication.

Since the parties P_i and R_i do not participate in the computation, they do not have to participate in the verification. However, they will still be punished if they provide multiple signatures for the same value.

5.4.2 Possible Issues

The main drawback of the previous proposition is the numerous amount of signatures that the computing parties may have to check. While in the initial scheme each party M_j has to provide just one share $\mathbf{x_{jkT_\ell}}$ for each party M_k in each T_ℓ, now it has to provide N shares, where N is the number of input parties, and all their signatures have to be checked (for M_j it is still sufficient to use just one signature, but it does not help much). Depending on the settings, N can be very large. In the worst case, each input party provides only one bit, and hence $N \in O(|C|)$. However, each computing party would have to verify the source of all the inputs anyway. For P_i, sending $t \cdot \binom{n-1}{t}$ shares instead of one is not worse since all the values used by the same P_i may have the same signature. The problem comes when M_j wants to redistribute the shares and the signatures to all T-sets, since each receiver will again have to check all N of them. Fortunately, this happens only in the beginning and in the end of the protocol.

Additionally, depending on the performed computation, the covert security may just not work with the input parties, especially in some anonymous statistical projects. Any participant may cheat without reason and complain afterwards. In our scheme, the verification of input share signatures is done already in the beginning, an hence the computing parties will not spend their time on clearly malicious parties whose shares do not correspond to their signatures. The problem still remains with the output, since the malicious output party R_i may sign wrong values just for fun, without fear of being detected. However, since such cheating would require just one additional broadcast (revealing the signatures to everyone), this is not too much different from the case if R_i has not complained. In any case, even if no one complains, it may still be some kind of attack where the input party is completely honest, but it just performs the computation without needing.

5.4.3 Deviations from the Initial Settings

In real Secure Multiparty Computation Platforms, it may happen that the number of input parties is initially unknown. For example, in the case of some statistical computation, the input parties may come and submit their inputs during the execution, and hence the shape of the computational circuit may be even unknown in the beginning, since the input length is undefined. Nevertheless, the proposed techniques still work. The coming input parties may commit the inputs as they come. In the end of the computation, the structure of the circuit will be known anyway.

5.5 Conclusions and Future Work

In this work we have proposed a scheme that allows to verify the computation of each party in a passively secure protocol, thus converting passive security to covert security. Each malicious party will be detected with probability close to 1, depending on the parameters of selected field.

While our verification is being done only after the entire computation has ended, it might be interesting to do something more similar to the active security model. Namely, we could require each party to prove the correctness after each round. If implemented straightforwardly, repeating our verification algorithm on each round, it multiplies the verification complexity by the number of rounds (actually, a bit less since in the beginning the vectors will be of smaller length). Doing it more cleverly, we could make use of the proofs of the previous rounds, making the next proof steps reliable on the proofs of the previous steps. The ideas can be taken for example from [14].

Chapter 6

Actively Secure Two-Party Computation with Precomputing

This chapter introduces two initialisations of actively secure two-party computation and new ideas for precoputation using additively homomorphic encryption. The main focus of secure computation in the precomputation model is usually on improving the efficiency of the online phase. However, the efficiency of the precomputation phase is also of importance. This work proposes ideas based on packing several values to one ciphertext to improve precomputation based on additive secret sharing. In addition, a symmetric and asymmetric setting for secret sharing are introduced where the symmetric gains most from the improved precomputation.

The aim of this work is to adapt the SPDZ general actively secure computation framework [23] for the two-party case and focus on optimising the precomputation phase. An important distinction between our work and SPDZ is that we use an additively homomorphic cryptosystem instead of the somewhat-homomorphic cryptosystem for the precomputation phase. We prefer an additively homomorphic cryptosystem because it is more conventional, easier to implement and currently more thoroughly studied. The resulting protocol set is implemented in Sharemind version 3 [46].

6.1 Related work

Secure computation is currently an active research field and has reached the state where is is efficient enough for practical applications. The work is mainly divided to three branches, one focusing on the development of garbled circuits, the second on secure multi-party computing on secret shared elements and the third on fully homomorphic encryption. This work is about the latter and considers secure computations on secret shared data. Sharemind is one of the more mature secure multi-party computation frameworks that currently offers *passive* security guarantees [11, 12]. This work uses the principles also combined in the SPDZ framework [23] to add an *actively* secure protocol set to the Sharemind framework.

Our protocols are divided between the *online* and *offline* world also known as the *precomputation* model. The precomputation model originates from Beaver [3] and has found wider usage in SMC after [21] as it has been used by [17, 22, 5, 40, 23]. Firstly, the precomputation phase is independent of the secret information and produces some random shares or sets of shares in a specific relation. Secondly, the online phase uses the secrets and the precomputation results to efficiently evaluate necessary functions.

The SPDZ framework utilises three important tools: oblivious message authentication codes (MAC) [44], Beaver triples [3], and vectorized homomorphic encryption [27, 47]. The first is used to ensure security against an active adversary and the second as a precomputation mechanism for multiplication. These two have been previously used together for SMC in BDOZ [5]. However, SPDZ adds an important idea that MAC is used to authenticate the shared secret as a whole and not for authenticating independent shares.

Vectorised somewhat-homomorphic encryption is used to generate Beaver triples in a communication-efficient way and is a SPDZ-specific property. Currently, SPDZ precomputes Beaver triples and single random shares. The covertly secure extension of SPDZ [19] also precomputes squaring pairs analogously to Beaver triples and shared bits for comparison, bit-decomposition, fixed point and floating point operations. It is an open question if other operations can be efficiently precomputed.

Our work also uses oblivious MAC and Beaver triples, but differently from SPDZ we use additively homomorpic Paillier cryptosystem [41] instead of fully homomorphic encryption.

6.2 Secure two-party computation

We require three things for secure computation that is secure against an active adversary: definition of the share, protection mechanisms to ensure the correctness of the computation results, and computation protocols. We use additive secret sharing and homomorphic message authentication as our main protection mechanism. In addition, the secret sharing method and homomorphic MAC should have the same operations that can be computed locally.

6.2.1 Possible setups

Frameworks analogous to SPDZ can be used with two computing parties and could have three considerably different initialisations. The main difference between them is the setup and means of using the MAC algorithm. Two parties are denoted by $\mathcal{CP}_1$ and $\mathcal{CP}_2$.

Asymmetric setup

Asymmetric setup differentiates the computing parties so that one gets the role of a master node ($\mathcal{CP}_1$) who defines the MAC key and the client ($\mathcal{CP}_2$) is using the keys from the master. Using the MAC to either authenticate the secret value or the share of the other party enables $\mathcal{CP}_1$ to easily verify the correctness of the declassification result. However, $\mathcal{CP}_2$ is unable to verify the MAC as it must not know the MAC secret key. It is up to the master to also define something that $\mathcal{CP}_2$ can check. For example, we can use commitments.

The MAC tag for the whole value or the share of $\mathcal{CP}_2$ can not be kept by the master node. MAC algorithms are not designed to protect the privacy of the message. Thus, seeing the whole tag might leak the secret to the master node, who also knows the MAC secret key. In addition, storing the tag on the side of $\mathcal{CP}_2$ might also leak some information about the secret or the key. Hence, we need to store the tag t in a secret-shared manner and both parties must be able to update their parts of the tags during the computation.

For our initialisation of the asymmetric protocol set, we use a MAC algorithm defined as $t = k \cdot x \bmod N$ for tag t, key k, Paillier modulus N and secret value x. In addition, we use additively homomorphic perfectly binding commitments based on the Paillier' cryptosystem that the party $\mathcal{CP}_2$ can verify. This results in the fact that all our computations will be with respect to the Paillier modulus.

Each secret value x is represented by a tuple

$$[\![x]\!]_N = \langle \Delta, x_1, x_2, r, (\![x_1]\!)_{pk}, z_1, z_2 \rangle$$

such that $x = x_1 + x_2 + \Delta \bmod N$ and $z_1 + z_2 = k \cdot (x_1 + x_2) \bmod N$. The values Δ and $(\![x_1]\!)_{pk} = \mathsf{Enc}_{pk}(x_1, r)$ are public whereas $\mathcal{CP}_i$ has private values z_i and x_i. The public modifier Δ is always 0 for random values and is used to enable fast addition of a share and public constant. Value r is kept by $\mathcal{CP}_1$ to open the commitment to $(\![x_1]\!)_{pk}$ of share $[\![x]\!]_N$. This randomness also enables us to write protocols so that actually only $\mathcal{CP}_2$ computes $(\![x_1]\!)_{pk}$ and $\mathcal{CP}_1$ recomputes the encryption if needed. This is a reasonable step because, in reality, $\mathcal{CP}_1$ rarely needs this value.

Symmetric setup

A symmetric setup means that both computing parties define similar parameters. A direct continuation of the previous asymmetric setting would be that both parties $\mathcal{CP}_1$ and $\mathcal{CP}_2$ in the symmetric setting define their own MAC keys k_i. This would mean that on top of the secret sharing method we have two MAC tags $z^{(1)}$, $z^{(2)}$ where both parties can verify one of them during the declassification phase. As in the asymmetric, case we need a to keep the tags in shares.

The main benefit of this setup over the asymmetric one is that the protocol descriptions would also become symmetric. This simplifies the notation and also means that the parties can do exactly the same workload in parallel. In some sense, this enables us to gain more efficient time usage. More precisely, it is unlikely that in such protocols one party has to wait between sending and receiving network message without having any computations to perform. Furthermore, we can only use the cheap MAC algorithm and do not need more expensive homomorphic commitments that we used in the asymmetric case.

For our initialisation, we use the same MAC as for the asymmetric case for both of the participants, but we use a different prime modulus p. We propose a share representation as

$$[\![x]\!]_p = \langle \Delta, x_1, x_2, z_1^{(1)}, z_2^{(1)}, z_1^{(2)}, z_2^{(2)} \rangle \ ,$$

where $x = x_1 + x_2 + \Delta \bmod p$ and Δ is the public modifier. The remaining values belong to the MAC tags as $z_1^{(i)} + z_2^{(i)} = k_i \cdot (x_1 + x_2) \bmod p$. Both parties know Δ and, in addition, $\mathcal{CP}_i$ has values x_i, $z_i^{(1)}$ and $z_i^{(2)}$.

Shared setup

The shared key model is a further extension changing the symmetric setup so that instead of both parties defining a key they share one key k between them. This defines a threshold MAC algorithm where all parties must participate in the verification of the tag. It can give additional efficiency gains as the parties only have to update a single tag t during the computations. However, the sharing of the key k is special as it has to define some additional information, allowing parties to verify the correctness of the restored key and checked tags. The shared key setup is the approach currently used by the SPDZ framework, however, we do not define our version of the shared setup.

There are well-known difficulties with shared approach as the knowledge of the secret key is usually needed to verify the MAC tags. One possible solution is to not verify any opened results before all computations are done. Afterwards, it is possible to restore the MAC key and verify all the results at once. However, in such case parties can only notice cheating very late and they must agree on a new key before next computations. In addition, changing the key means that after verifying the correctness of opened values, the shares of the outputs or intermediate results from the checked computations can not be reused.

6.2.2 Protocols

The description of some SPDZ protocols is independent from the secret sharing method as long as the scheme defines protocols for publishing shares privately to each computing or result party, generating a random share, and generating random Beaver triples. There are three main protocols: classifying the secret input, publishing the secret shared value, and multiplying two shared values. Classifying and multiplication protocols depend on the precomputation protocols. Publishing is actually dependant on the share representation, but the overall idea remains the same - open the value and verify that the protection mechanisms hold. We use these general protocols, but in addition have to define specific protocols for our share representation. In addition, we use a common protocol for verifying the correctness of Beaver triples from [22].

Our protocols include the addition protocol for the online phase and random share and triple generation protocols for the precomputation phase. The addition protocol is in a sense trivial, because we define our share representation so that the addition can be done locally. For precomputation we need protocols that generate either single random shares or random multiplicative triples. The main idea of the latter is that we at first generate two random shares and then use some multiplication functionality to obtain the third triple element. The main drawback is that we can not use the previously mentioned multiplication protocol because it requires triples as input. Therefore we need special protocols for *slow multiplication* that are discussed afterwards. All our protocols are fully specified in [42].

For practical computations, we would also need additional protocols. For example to compute division or exponentiation. However, addition and multiplication, together with the supporting protocols to work with secret sharing and precomputation, are sufficient for testing the feasibility of the proposed framework.

6.3 Beaver triple generation

In Beaver triple generation, we focus on the following question. Given two random elements a and b, we want to find their product. The main algorithm that we can use is a basic additive share multiplication using the Paillier cryptosystem. However, there is one problem with this algorithm, namely that it gives correct results for the Paillier modulus, but we might want to use different moduli for our triples. The other limitation is that if we are interested in very short a and b values compared to the Paillier modulus, then the encryption plaintext will have a lot of unused bits and hence, the computations are not very efficient. We try to overcome the first problem by introducing error correction and the second by packing several elements into one ciphertext.

6.3.1 Packing

B-ary packing

Packing as B-ary numbers means that each element modulo $M < B$ represents a digit and we pack them as numbers of base B. A three-digit base-B number could be written out as

$$x = B^2 \cdot x_3 + B \cdot x_2 + x_1 \enspace ,$$

where $x_i < B$ are digits. If we assume, that y is written out in a similar manner, then the corresponding multiplication becomes a degree 4 polynomial of B where we can learn x_3y_3 and x_1y_1, assuming that these do not overflow a B-ary digit.

Packing as straightforward B-ary numbers is, therefore, not very beneficial, as we did not receive a triple x_2, y_2, x_2y_2. However, we could consider another example, with x as before, but y is modified, giving $y = B^6y_3 + B^3 \cdot y_2 + y_1$. After multiplying it out, xy contains all the triples x_i, y_i and x_iy_i, but also some elements x_iy_j, $i \neq j$ that we do not need. Thus, in this packing we only get the same number of triples as the square-root of the number digits in the base-B representation of xy where the maximal size is limited by the plaintext size and, therefore, it is not very space-efficient.

Another problem with using this approach in a straightforward manner is that we have to assume that $y_ix_i < B$, which essentially means that the result xy contains integer results of all triples as digits. However, the common version of the Paillier multiplication would destroy this structure in the outcome. This packing can be used with a redefinition of the randomiser in the Paillier multiplication, but it will also result in decreased security. A version of partial B-ary packing was also introduced in [43], that also suffers from the reduced security level.

Packing using the Chinese Remainder Theorem

The Chinese remainder theorem (CRT) can also be used for packing several elements into one ciphertext. However, it can only be used, if we are using elements with pairwise coprime moduli p_i. By definition, CRT can be used to combine all those single random values x_i modulo p_i, for a modulus $M = p_1 \cdot \ldots \cdot p_k$ and execute the triple generation protocol to obtain the corresponding third triple element xy modulo M. We are interested in learning the shares for x_iy_i. The CRT allows us to reduce the final result xy respectively for all moduli p_i to learn the third triple element for all initial random value pairs. Therefore, we can learn x_iy_i from $xy \bmod M$ as $x_iy_i = xy \bmod p_i$. Packing with CRT enables us to get exactly $|M|$-bit triples from one execution of the triple generation protocol where the maximal size of M is bounded by the used Paillier modulus.

However, in most use-cases, we would like to always use the same modulus p, not different p_i. Actually, we could use a modulus $p < p_i$ and convert the final results for different moduli p_i to one modulus p using the ideas from the following error correction section.

6.3.2 Error correction

Error correction is the most important part of the share conversion step. Here, share conversion means that we have a secret value x shared using a modulus M_1 and we need to obtain a sharing of x for a modulus M_2. In addition, we are only interested in cases where $M_2 \leq M_1$ and M_1 is odd, because these are what we need for using Paillier multiplication or packing with the Chinese remainder theorem. Especially, we have some $x = a \cdot b \bmod M_1$ and we need to obtain $x = a \cdot b \bmod M_2$. By picking a and b, we can actually ensure that $a \cdot b \leq M_1$, which means that the value x can be reduced as usual.

The second problem is that we are working with secret sharing either modulo M_1 or M_1. Initially, we have shared value $x = x_1 + x_2 \bmod M_1$ with shares x_1 and x_2 and we actually need to learn shares x_1^* and x_2^* such that $x = x_1^* + x_2^* \bmod M_1$.

There are two possibilities, $x_1 + x_2 < M_1$ or $2M_1 > x_1 + x_2 \geq M_1$. If we are in the first case, then $x_i^* = x_i \bmod M_2$. However, for the second case $x_1^* = x_1 \bmod M_2$, but $x_2^* = x_2 - M_1 \bmod M_2$. These cases can be distinguished easily using only the least significant bits of x, x_1 and x_2, because if $x \bmod 2 = x_1 + x_2 \bmod 2$ then we are in the first case and otherwise in the second. Moreover, this condition can be securely checked using oblivious transfer, if we know what the last bit of x should be. We can use this, for example, to instead check the value $2x$, where we know that the least significant bit is 0.

The main drawback of using $2x$ as a check value is that in such a case, we can not use modulus 2 as any of our target moduli. However, it is usable for most use-cases, especially for transforming the Paillier modulus to a prime modulus needed for the symmetric and shared versions of secure computation.

Finally, it would also be possible that instead of doing error correction we use values where the probability of an error is small enough and use the triple verification procedure to check that the triple was indeed valid.

6.4 Conclusion

Current results show that actively secure multi-party computation is significantly slower than passively secure versions. However, our results indicate that fully implemented symmetric protocol set could be close to the performance of the SPDZ framework that is the current leader in actively secure multi-party computation frameworks. In addition, achieving security against malicious adversaries can be very important for data mining tasks that have important economical or societal outcomes. Therefore, in many cases the extra time consumption is a reasonable trade-off for the additional layer of security.

In conclusion, the symmetric and possibly shared setups are the most reasonable setups for secure two-party computation. Follow up work should focus more on specifying the missing details of the symmetric setup and on achieving the setup phase of the shared setup.

Chapter 7

Comparison of oblivious sorting algorithms

7.1 Introduction

Sorting is an important operation in privacy-preserving data analysis and data mining. In addition to its obvious use in ordering data, sorting is used for finding ranked elements (top-k, quantiles), performing group-level aggregations and implementing statistical tests.

In UaESMC deliverable D2.2.1 [8] we showed how sorting networks can be evaluated in SMC. In this chapter, we concentrate on constructions based on oblivious shuffling and introduce further optimizations for sorting networks.

7.2 Oblivious sorting techniques

7.2.1 Constructions based on comparisons

Comparison-based sorting algorithms use the comparison operation to determine the correct sequence of elements of a given array. Such algorithms are inherently data-dependent, as the execution flow depends on the outcomes of the comparison operations. Hence, changes in the input data will affect the running time of the algorithm. A simple solution would be to evaluate all the branches of the sorting algorithm and obliviously select the correct output in the end, but this dramatically reduces efficiency.

Hamada *et al.* [28] propose a generic solution to *obliviously shuffle* [36] the inputs before performing a comparison-based sort. Then, as we are comparing values in a randomly shuffled vector, any declassified (published) comparison results are also random. However, the pattern of comparisons in the algorithm can still leak information, such as the number of equal elements in the input vector.

Because many SMC implementations have highly efficient vector operations, vectorized naive protocols may sometimes be more efficient than protocols with a lower computational complexity and a lower degree of vectorization. In this paper, we propose a naive sorting protocol based on shuffle and vectorized comparisons called NaiveCompSort (Algorithm 10). In this algorithm, we first shuffle the input array and then compare every element with every other element in the array in one big vector operation. Finally, we rearrange the elements according to the declassified comparison results. This algorithm always works in the worst case time of $\mathcal{O}(n^2)$ and its runtime is, therefore, data-independent.

7.2.2 Constructions specific for bitwise secret-sharing schemes

If data is secret-shared using a bitwise secret-sharing scheme, access to individual bits is cheap. This allows us to design a very efficient count/radix sorting algorithm. Counting sort [15, 24] is a sorting algorithm that can sort an array of integers in a small range by first constructing a frequency table and then rearranging items in the array according to this table. Algorithm 11 describes a counting sort algorithm for binary data.

Algorithm 10: NaiveCompSort

```
   Data: Input array $[\![\mathcal{D}]\!] \in \mathbb{Z}_{2^k}^n$
   Result: Sorted array $[\![\mathcal{D}']\!]$
 1 Let $[\![T]\!] = \mathsf{Shuffle}([\![T]\!])$
   // All comparisons here are done in parallel.
 2 for $i < j \in \{1, 2, \ldots, n\}$ do
 3     Let $[\![g_{i,j}]\!] = [\![\mathcal{D}_i]\!] \leq [\![\mathcal{D}_j]\!]$
 4 end
 5 Declassify the values $[\![g_{i,j}]\!]$ and sort $[\![\mathcal{D}]\!]$ according to them, obtaining $[\![\mathcal{D}']\!]$
 6 return $[\![\mathcal{D}']\!]$
```

Algorithm 11: Counting sort algorithm for binary arrays.

```
   Data: Binary input array $\mathcal{D} \in \mathbb{Z}_2^n$.
   Result: Array $\mathcal{D}' \in \mathbb{Z}_2^n$ with elements of $\mathcal{D}$ in increasing order.
 1 $n_0 \leftarrow n - sum(\mathcal{D})$; // Count number of zeros.
 2 $c_0 \leftarrow 0$; $c_1 \leftarrow 0$; // Keep counters for processed zeros and ones.
   // Put each element in right position:
 3 foreach $i \in 1 \ldots n$ do
 4     if $\mathcal{D}_i == 0$ then
 5         $c_0 = c_0 + 1$
 6         $\mathcal{D}'_{c_0} = \mathcal{D}_i$
 7     else
 8         $c_1 = c_1 + 1$
 9         $\mathcal{D}'_{n_0+c_1} = \mathcal{D}_i$
10     end
11 end
12 return $\mathcal{D}'$
```

Radix sort [30] sorts an array of integers by rearranging them based on counting sort results on digits in the same positions. Radix sort sorts data one digit position at a time, starting with the least significant digit. This works as the underlying counting sort is a stable sorting algorithm. Algorithm 12 shows the full protocol of oblivious radix sort that uses binary counting sort as a subroutine. The underlying counting sort is made data-independent by obliviously updating counters $[\![c_0]\!]$, $[\![c_1]\!]$ and the order vector $[\![ord]\!]$. Such a data-independent counting sort is sufficient to make our radix sort data-independent as well.

As our radix sort algorithm does not use comparison operations, it is not bound by the computational complexity lower bound of $\Omega(n \log n)$ for comparison-based sorting algorithms. Counting sort has a complexity of $\mathcal{O}(n)$ and radix sort on k-digit elements that uses counting sort as a subroutine, has a computational complexity of $\mathcal{O}(kn)$. However, the data-independent counting sort protocol also uses addition and multiplication operations which are expensive protocols on bitwise shared data. Therefore, after creating a vector with bits on a given position, we convert it to additively shared data and work in this domain. The output of the algorithm is still in a bitwise form.

7.3 Optimization methods and matrix sorting

7.3.1 Vectorization

In their work, Hamada *et al.* parallelize the invocations of comparisons on secret-shared data as network communication is the main bottleneck for SMC protocols. Such SIMD (single instruction, multiple data) operations are very efficient as they allow to put messages of many parallel operations into a single network

Algorithm 12: Data-independent radix sort.

```
   Data: Input array $[\![\mathcal{D}]\!] \in \mathbb{Z}^n_{2^k}$.
   Result: Sorted array $[\![\mathcal{D}]\!] \in \mathbb{Z}^n_{2^k}$.
   // Iterate over all digits starting with the least significant digit:
 1 foreach $m \in 1 \ldots k$ do
       // Construct a binary vector consisting of $m$-th digits.
       // Convert it to additively shared data.
 2     $[\![d]\!] \leftarrow \mathsf{ShareConv}(([\![\mathcal{D}_1]\!]_m, [\![\mathcal{D}_2]\!]_m, \ldots, [\![\mathcal{D}_n]\!]_m))$
 3     $[\![n_0]\!] \leftarrow n - sum([\![d]\!])$; // Count number of zeros.
 4     $[\![c_0]\!] \leftarrow 0$; $[\![c_1]\!] \leftarrow 0$; // Keep counters for processed zeros and ones.
 5     $[\![ord]\!]$; // Keep $n$-element shared order vector.
       // Put each element in the right position:
 6     foreach $i \in 1 \ldots n$ do
 7         $[\![c_0]\!] = [\![c_0]\!] + 1 - [\![d_i]\!]$
 8         $[\![c_1]\!] = [\![c_1]\!] + [\![d_i]\!]$
           // Obliviously update order vector:
 9         $[\![ord_i]\!] = (1 - [\![d_i]\!]) * [\![c_0]\!] + [\![d_i]\!] * ([\![n_0]\!] + [\![c_1]\!])$
10     end
11     $([\![\mathcal{D}]\!], [\![ord]\!]) \leftarrow \mathsf{Shuffle}([\![\mathcal{D}]\!], [\![ord]\!])$; // Shuffle two column database.
12     $ord \leftarrow \mathsf{Declassify}([\![ord]\!])$
13     Rearrange elements in $[\![\mathcal{D}]\!]$ according to $ord$.
14 end
15 return $[\![\mathcal{D}]\!]$
```

message, saving on networking overhead. They did not only parallelize comparison invocations in a single partitioning subroutine, but rather all the comparisons at each depth of the quicksort algorithm. For this work we implemented the oblivious quicksort algorithm proposed in [28] as a reference, using the same idea for parallelization.

Similarly, we vectorize all secure operations in our counting sort algorithm design. As sorting by a given digit position is dependent on the previous position outcome, radix sort cannot be vectorized further. We could apply counting sort on chunks of 2 or more bits and reduce the number of rounds for radix sort. However, this requires substituting the cheap oblivious choice subprotocol for a more expensive comparison protocol.

7.3.2 Changing the share representation

Both comparison-based sorting algorithms and sorting networks rely on the comparison operation. Comparison is a bit-level operation and works faster on bitwise shared data. Therefore, we can convert additively shared inputs into bitwise shared form and run the intended algorithm on the converted shares. The results can be converted back to additively shared form at the end of the algorithm.

Converting additive shares to bitwise shares requires a bit extraction protocol. However, for algorithms that perform many comparisons after one another, the benefits of many fast comparisons outweigh one costly conversion.

7.3.3 Optimizations specific to sorting networks

In software implementations, the generation of sorting networks can take a significant amount of time. As the sorting network structure is data-independent, we can store the sorting network after generation to re-use it later.

If we shuffle the inputs before sorting, we can optimize the CompEx function implementations by declassifying comparison results and performing the exchanges non-obliviously. The running time of the resulting algorithm is data-independent because of the constant structure of the sorting network.

7.3.4 Sorting matrices

Sorting secret shared matrices using sorting networks is covered in UaESMC deliverable D2.2.1 [8].

Similarly, shuffle-based algorithms can be easily modified to support matrix sorting. Assume that our input data is in the form of a matrix $\mathcal{D}_{i,j}$ where $i = 1 \dots n$ and $j = 1 \dots m$. Let us also fix a column k by which we want to sort the rows.

First, we obliviously shuffle the rows in the whole matrix. Note that shuffling is already a part of comparison-based sorting protocols like quicksort and NaiveCompSort. However, this extra step has to be added for radix sort. Next, we extract the k-th column from the matrix and pass it to the sorting algorithm of our choice together with an n-element index vector $(1, 2, \dots, n)$.

The sorting protocol now swaps elements in the data vector and the index vector together. After sorting these two vectors, we declassify the output index vector and use it as a permutation to rearrange rows in the matrix. Declassifying the index vector leaks information on how the elements were rearranged. However, as the input matrix was obliviously shuffled, this leaks no information on the original placement of rows in the initial matrix.

7.4 Conclusion

We describe three designs for oblivious versions of known sorting algorithms—naive comparison-based sort, quicksort and radix sort. As opposed to sorting networks, all of these perform some declassifications to improve efficiency. Our novel oblivious radix sorting algorithm leaks less information than constructions based on shuffling and declassified comparison results.

In addition, we recommend to use precomputing or caching of network structure for oblivious sorting networks. In that case, the algorithm provides perfect privacy with a reasonable performance.

Bibliography

[1] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 681–698. Springer, 2012.

[2] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.

[3] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[4] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *CRYPTO (2)*, pages 90–108, 2013.

[5] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[6] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, pages 315–333, 2013.

[7] 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.

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

[9] 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.

[10] Dan Bogdanov, Liina Kamm, Sven Laur, and Pille Pruulmann-Vengerfeldt. Secure multi-party data analysis: end user validation and practical experiments. Cryptology ePrint Archive, Report 2013/826, 2013. `http://eprint.iacr.org/`.

[11] 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.

[12] 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.

[13] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[14] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In Andrew Chi-Chih Yao, editor, *ICS*, pages 310–331. Tsinghua University Press, 2010.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 8.2 Counting Sort, pages 168–170. MIT Press and McGraw-Hill, 2nd edition, 2001.

[16] 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.

[17] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[18] Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. From passive to covert security at low cost. In Daniele Micciancio, editor, *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010.

[19] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[20] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In Yuval Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2011.

[21] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590. Springer, 2007.

[22] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 558–576. Springer, 2010.

[23] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[24] Jeff Edmonds. *How to Think about Algorithms*, chapter 5.2 Counting Sort (a Stable Sort), pages 72–75. Cambridge University Press, 2008.

[25] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[26] 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.

[27] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[28] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *Proc. of ICISC'12*, volume 7839 of *LNCS*, pages 202–216. Springer, 2013.

[29] Myles Hollander and Douglas A Wolfe. *Nonparametric statistical methods.* John Wiley New York, 2nd ed. edition, 1999.

[30] Herman Hollerith. US395781 (A) - ART OF COMPILING STATISTICS. European Patent Office, 1889. `http://worldwide.espacenet.com/publicationDetails/biblio?CC=US&NR=395781`.

[31] Rob J Hyndman and Yanan Fan. Sample quantiles in statistical packages. *The American Statistician*, 50(4):361–365, 1996.

[32] Peeter Laud and Alisa Pankova. New Attacks against Transformation-Based Privacy-Preserving Linear Programming. In Rafael Accorsi and Silvio Ranise, editors, *Security and Trust Management (STM) 2013, 9th International Workshop*, volume 8203 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2013.

[33] Peeter Laud and Alisa Pankova. On the (Im)possibility of Privately Outsourcing Linear Programming. In Ari Juels and Bryan Parno, editors, *Proceedings of the 2013 ACM Workshop on Cloud computing security, CCSW 2013*, pages 55–64. ACM, 2013.

[34] Peeter Laud and Alisa Pankova. Verifiable computation in multiparty protocols with honest majority. Cryptology ePrint Archive, Report 2014/060, 2014. `http://eprint.iacr.org/`.

[35] Peeter Laud and Jan Willemson. Universally composable privacy preserving finite automata execution with low online and offline complexity. Cryptology ePrint Archive, Report 2013/678, 2013. `http://eprint.iacr.org/`.

[36] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*, pages 262–277, 2011.

[37] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.

[38] Helger Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. *IACR Cryptology ePrint Archive*, 2013:121, 2013.

[39] Helger Lipmaa and Tomas Toft. Secure equality and greater-than tests with sublinear online complexity. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *ICALP (2)*, volume 7966 of *Lecture Notes in Computer Science*, pages 645–656. Springer, 2013.

[40] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. *CoRR*, abs/1202.3052, 2012.

[41] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[42] Pille Pullonen. Actively secure two-party computation: Efficient Beaver triple generation. Master's thesis, University of Tartu, Aalto University, 2013.

[43] Pille Pullonen, Dan Bogdanov, and Thomas Schneider. The design and implementation of a two-party protocol suite for Sharemind 3. Technical report, Cybernetica AS Infoturbeinstituut, 2012. `http://research.cyber.ee`.

[44] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In David S. Johnson, editor, *STOC*, pages 73–85. ACM, 1989.

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

[46] Sharemind. `http://sharemind.cyber.ee`. Last accessed 2013-10-10.

[47] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *IACR Cryptology ePrint Archive*, 2011:133, 2011.

[48] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.