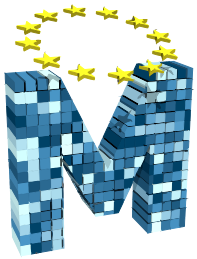


Project title: Multi-Owner data Sharing for Analytics and Integration respecting Confidentiality and OWNer control
Project acronym: MOSAICrOWN
Funding scheme: H2020-ICT-2018-2
Topic: ICT-13-2018-2019
Project duration: January 2019 – December 2021

D5.4

Final Versions of Tools for Data Sanitisation and Computation

Editors: Stefano Paraboschi (UNIBG)
Dario Facchinetti (UNIBG)
Gianluca Oldani (UNIBG)
Matthew Rossi (UNIBG)
Reviewers: Matthew Keating (EISI)
Rigo Wenning (W3C)

Abstract

This document reports on the implementation efforts associated with the development of two tools supporting novel techniques for data sanitization. The proposed approaches specifically focus on scenarios where different parties aim at collaborating to anonymize a dataset or to compute (in a privacy preserving manner) statistics over private data. The document first illustrates an anonymization approach that operates in a distributed scenario. The proposal specifically extends the Mondrian algorithm to leverage the presence of multiple workers to improve scalability and enable the efficient anonymization of large data collections. The developed tool is able to compute a k-anonymous and ℓ-diverse version of the dataset relying on an arbitrary number of workers, without affecting information loss. The tool has received the Best Artifact Award at the IEEE PerCom 2021 Conference.

The document then presents a novel solution for computing the differentially private median over the union of two private datasets. The performance of the developed solution outperforms existing approaches, while maintaining utility comparable to computations performed in a centralized scenario.

Type	Identifier	Dissemination	Date
Deliverable	D5.4	Public	2021.09.30

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825333.

MOSAICrOWN Consortium

1.	Università degli Studi di Milano	UNIMI	Italy
2.	EMC Information Systems International	EISI	Ireland
3.	Mastercard Europe	MC	Belgium
4.	SAP SE	SAP SE	Germany
5.	Università degli Studi di Bergamo	UNIBG	Italy
6.	GEIE ERCIM (Host of the W3C)	W3C	France

Versions

Version	Date	Description
0.1	2021.09.06	Initial Release
0.2	2021.09.27	Second Release
1.0	2021.09.30	Final Release

List of Contributors

This document contains contributions from different MOSAICrOWN partners. Contributors for the chapters of this deliverable are presented in the following table.

Chapter	Author(s)
Executive Summary	Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG)
Chapter 1: Scalable Distributed Data Anonymization	Sabrina De Capitani di Vimercati (UNIMI), Dario Facchinetti (UNIBG), Sara Foresti (UNIMI), Gianluca Oldani (UNIBG), Stefano Paraboschi (UNIBG), Matthew Rossi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 2: Secure Computation of Differentially Private Statistics	Jonas Böhler (SAP SE)
Chapter 3: Conclusions	Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG)

Contents

List of Figures

List of Tables

Executive Summary

This deliverable presents two solutions for protecting sensitive data through data sanitization developed in WP5, and their implementation. The amount of data generated and collected on a daily basis is continuously growing (e.g., sensors constantly collect information about ourselves and the environment where we live). Such data are valuable and may need to be shared with others without, however, violating the privacy of the individuals to whom they refer. Also, the different parties collecting the data might require their data sets to remain private, while interested in computing statistics over their union. This deliverable specifically considers distributed scenarios, characterized by different interacting parties, aimed at collaborating for computing an anonymized version of the collected data or privacy-preserving statistics over these data.

The document is organized in two chapters, each describing an implementation effort toward the support of complementary approaches to data sanitization. The current state of the art shows that applications need both on one side anonymization based on the controlled release of (quasi-) identifying information, and on the other side approaches based on differential privacy.

Chapter 1 illustrates an extended version of Mondrian algorithm that operates in a distributed environment, parallelizing the anonymization task among an arbitrary number of workers. The chapter also presents a tool implementing our distributed version of the Mondrian algorithm, and experimental results proving its scalability and the quality of the anonymized dataset.

Chapter 2 addresses the problem of computing the median over private datasets under the control of different parties. The proposed solution securely computes a differentially private median that does not expose sensitive information about the original datasets, using the exponentiation mechanism. The chapter illustrates the proposed computation protocol, which operates in time sublinear in the size of the data universe while not affecting utility. We note that the developed protocol is flexible, and can be adapted to compute other rank-based statistics in a differentially private manner.

1. Scalable Distributed Data Anonymization

In this chapter, we present an approach for enabling a distributed anonymization process over large collections of data and the tool implementing it. Our approach anonymizes large datasets (which might not fit in main memory) using an arbitrary number of workers within the Spark framework. We describe how to parallelize the anonymization process through a proper partitioning of the dataset. Our experimental evaluation shows that the proposed approach is scalable and does not affect the quality of the anonymized dataset.

The tool described in this chapter is available as open source in the Git repository that can be accessed at this URL: *https://github.com/mosaicrown/mondrian* [Rep]. The tool has been submitted as an artifact to the IEEE PerCom 2021 conference, where it received the Best Artifact Award.

1.1 Basic Concepts

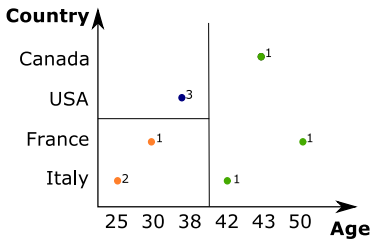
Guaranteeing privacy in datasets containing possible identifying and sensitive information requires not only refraining from publishing explicit identities, but also obfuscating data that can leak (disclose or reduce uncertainty of) such identities as well as their association with sensitive information. k-anonymity [CDFS07, DFLS12, Sam01], extended with ℓ-diversity [MGK06], offers such protection. k-anonymity requires generalizing values of the *quasi-identifier* attributes (i.e., attributes that leak information on respondent's identities by exploiting linkage with external sources) to ensure each quasi-identifier combination of values to appear at least k times. ℓ-diversity considers each sensitive attribute in such operation so to ensure each combination of quasi-identifier values to be associated with at least ℓ different values of the sensitive attribute (see Figure 1.1(c)).

While simple to express, k-anonymity and ℓ-diversity are far from simple to enforce, given the need to balance privacy (in terms of the desired k and ℓ) and utility (in terms of information loss due to generalization). Also, the computation of an optimal solution requires evaluating (based on the dataset content) which quasi-identifying attributes generalize and how, and hence demands complete visibility of the whole dataset for operating the generalization steps. Hence, existing solutions implicitly assume to operate in a centralized environment. Such an assumption clearly does not fit pervasive systems where the amount of data collected is huge. While scalable distributed architectures can help in performing computation on such large datasets, their use in computing an optimal k-anonymous solution requires careful design. In fact, a simple distribution of the load among workers would affect either the quality of the solution or the scalability of the computation (requiring expensive synchronization and data exchange among workers [AKS21]).

In this chapter, we demonstrate our scalable, efficient, and effective approach for the distributed enforcement of k-anonymity and ℓ-diversity requirements on large datasets. Our solution is based on an adaptation of Mondrian [LDR06], revised to operate without requiring knowledge

Age	Country	TopSpeed
25	Italy	132
25	Italy	132
30	France	128
42	Italy	110
50	France	115
43	Canada	115
38	USA	126
38	USA	127
38	USA	140

(a)



(b)

Age	Country	TopSpeed
[25-30]	Europe	132
[25-30]	Europe	132
[25-30]	Europe	128
[42-50]	World	110
[42-50]	World	115
[42-50]	World	115
38	USA	126
38	USA	127
38	USA	140

(c)

Figure 1.1: An example of a dataset (a), its spatial representation and partitioning (b), and a 3-anonymous and 2-diverse version (c), considering quasi-identifier `Age` and `Country` and sensitive attribute `TopSpeed`

of the complete dataset. Mondrian is a multi-dimensional algorithm that has established itself as an efficient and effective approach for achieving k-anonymity. Mondrian leverages a spatial representation of the data, mapping each quasi-identifier attribute to a dimension and each combination of values of the quasi-identifier attributes as a point in such a space. Mondrian then recursively cuts the tuples in each partition (the whole dataset at the first step) based on their values (lower/higher than the median) for a quasi-identifying attribute chosen at each cut. The algorithm terminates when any further cut would generate sub-partitions with less than k tuples, at which point values of the quasi-identifier attributes in a partition are substituted with their generalization. Figure 1.1(b) shows the spatial representation and partitioning of the dataset in Figure 1.1(a), where the number associated with each data point is the number of tuples with such values for the quasi-identifier in the dataset.

We have extended Mondrian designing a solution for partitioning data for distribution to workers without requiring knowledge of the whole dataset. We have implemented such an approach providing parallel execution on a dynamically chosen number of workers. The design of our partitioning approach aims at limiting the need for workers to exchange data, by splitting the dataset into as many partitions as the number of workers, which can independently run a revised version of Mondrian on their portion of the data. The experimental evaluation shows that our solution provides scalability, while not affecting the quality of the computed solution.

1.2 Distributed Anonymization

We illustrate the architecture and working of our system, supporting the distributed anonymization of large datasets.

1.2.1 Architecture

Figure 1.2 illustrates the architecture of our system, which includes two clusters: a *Hadoop Distributed File System* (HDFS) *cluster*, a well known and widely used solution for data storage and management, and a *Spark cluster* for data processing. Data is split in smaller blocks stored at *datanodes*. A *namenode* in the HDFS cluster manages the data stored at the datanodes and the access requests to them. For data processing, we have opted for *Spark* because it is a widely used

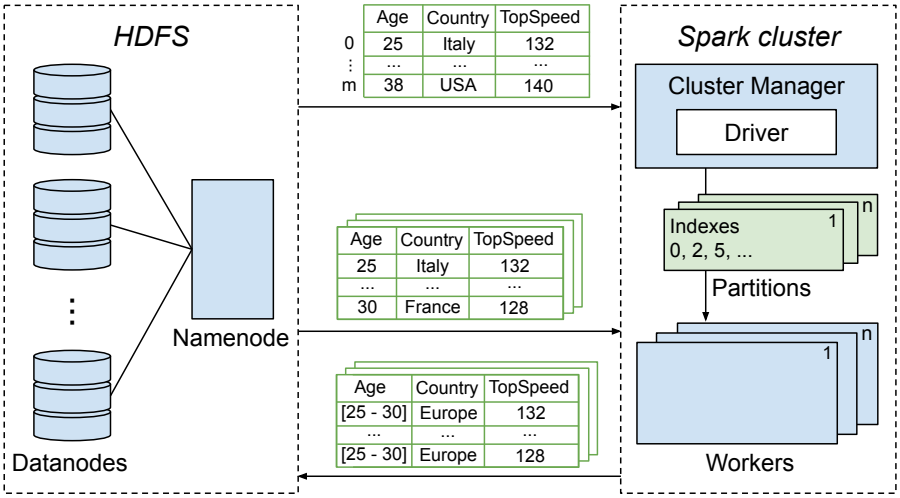


Figure 1.2: Architecture and working of the distributed anonymization system

engine for big data analytics that is fully compatible with the HDFS cluster. Among the nodes in the Spark cluster, one acts as *Spark Cluster Manager* and coordinates the work of the other nodes in the cluster, acting as *workers*.

Our distributed SPARK anonymization application has been developed in Python to leverage the Pandas framework, which can be conveniently used for managing large data collections. The application is associated with a *Spark Driver*. The Spark Driver, which runs on the Spark Cluster Manager, is responsible for converting the application into a set of jobs that are then divided into smaller execution units, called tasks. The tasks are allocated to workers by the Spark Cluster Manager.

1.2.2 Distributed Anonymization Algorithm

Our application operates in three steps (Figure 1.2): *pre-processing*, which partitions the dataset and distributes tasks to workers; *anonymization*, which anonymizes the dataset; *wrap-up*, which computes the information loss and collects other information related to the anonymization process.

Pre-processing. The first problem addressed consists in deciding how the dataset can be partitioned by the Spark Driver among the n available workers, in such a way that each worker can independently apply the anonymization algorithm on the portion of data assigned to it, without incurring in too much information loss. We first observe that, while a random partitioning of the dataset would work, it may increase the information loss. We therefore apply a strategy similar to the strategy used by the original Mondrian algorithm for creating sub-partitions: we first select an attribute of the quasi-identifier on which to partition the dataset and then create n partitions (one for each worker) depending on the values of the selected attribute. The attribute can be selected by applying different metrics (our tool supports maximum entropy, minimum entropy, and maximum span) that require to have the dataset in main memory to determine the distribution of the quasi-identifying attributes' values. To overcome this problem, we operate on a *sample* of the dataset (whose size is a configuration parameter of our tool) that fits into the main memory of the Spark Driver. Based on the randomly extracted sample, the Spark Driver determines the most suitable attribute, and partitions the tuples in the dataset according to the n-quantiles. We note that, as con-

firmed by the experimental results (Section 1.4), operating on a sample of tuples for performing the first partitioning of the dataset does not affect the quality of the solution.

Anonymization. The Spark Cluster Manager assigns the task of anonymizing each partition determined in the pre-processing step to a worker, depending on different factors (e.g., the workload, the datanode where data is stored). To make the system scalable, our implementation forces each partition to be assigned to a different worker. Each worker then downloads from the HDFS datanodes its portion of the dataset, and runs a revised version of Mondrian, without the need of interacting with the other workers. Our revised version of Mondrian differs from the original algorithm in two aspects: *1)* the attribute selected for partitioning is determined by applying the same metric used in the pre-processing step; *2)* the partitioning is performed considering both the k-anonymity and ℓ-diversity requirements. When the partitions cannot be further divided without violating k-anonymity nor ℓ-diversity, the tuples in each partitions are generalized. Our tool implements different generalization strategies, suited for different kinds of data (e.g., ranges for numeric attributes, user-defined generalization hierarchies for categorical attributes). Before storing the anonymized portion of dataset back at the datanodes, each worker computes the information loss on its portion of the dataset and sends the result to the Spark Driver (see next step).

Wrap-up. To assess the quality of the anonymized dataset, the Spark Driver computes the information loss produced by our distributed anonymization algorithm. To this end, the Spark Driver combines the values of the information loss received from the workers. Such a combination is done depending on the information loss metric adopted. Our tools support two of the most common metrics, that is, the Discernibility Penalty (DP) and the Global Certainty Penalty (GCP) [XWP+06].

1.3 Implementation

In this section, we illustrate the implementation of the proposal discussed in Section 1.2, describing the hardware and software requirements, the deployment of the tool and how to use it.

1.3.1 Hardware and Software Requirements

Hardware requirements. The deployment of the tool requires a machine having:

- a CPU with at least one logical core for each worker;
- at least 2 GB of RAM for each worker.

Software requirements. The deployment of the tool requires a machine with Linux operating system (the experimental results have been obtained using Ubuntu 20.04 LTS) and the following packages installed:

- *make*, version 4.3;
- *git*, version 2.27.0;
- *zip*, version 3.0, and *gzip*, version 1.10-2;
- *python3*, version 3.8.6;
- *python3-venv*, version 3.8.6;

- *gnuplot*, version 5.2 patchlevel 8.

When these packages are available, the environment set up should be finalized through the following steps:

1. install and set up *docker* and *docker-compose* (for more details on this step, see Section "Prerequisites" at [Rep]);
2. run `sudo usermod -aG docker <USER>`;
3. reboot the system;
4. check that the following commands run without root privileges:
   `docker run hello-world`
   `docker-compose -version`

1.3.2 Deployment of the Tool

The steps for deploying the tool are the following:

1. clone the repository through command

   ```
   git clone --depth 1 --branch \
   percom2021_artifact \
   https://github.com/mosaicrown/mondrian.git
   ```

2. run `make` to verify that all the software requirements illustrated in Section 1.3.1, which are needed for the distributed (Spark-based) version of the algorithm, are satisfied by the environment;
3. run `make start` to pull and build a copy of the Docker images necessary to the tool.

The tool uses the following Docker containers:

- *Hadoop Namenode* at `http://localhost:9870`
  (the web page available at this url permits to check the status of the *Hadoop Datanode* and to browse the distributed file system);
- *Hadoop Datanode* at `http://localhost:9864`;
- *Spark History Server* at `http://localhost:18080`;
- *Spark Cluster Manager* (and thus *Spark worker*s) at
  `http://localhost:8080`.

1.3.3 Use of the Tool

The tool implements the centralized and our distributed version of the Mondrian algorithm. The tool is complemented with a web UI that can be deployed running command `make ui`. The web UI is available at `http://localhost:5000` and can be used to run customized experiments. A complete user guide to the web UI is available at [Rep]. In the following, we describe the use of the tool to reproduce the experimental results presented in Section 1.4.

IPUMS USA dataset. The experiments presented in Section 1.4 used a sample from the IPUMS USA dataset [RFG+20]. The dataset is available at `https://ipums.org/`, together with a detailed guide for its download. To extract the sample to anonymize from the same dataset used in our experiments, go to IPUMS website `https://usa.ipums.org/usa/` and click on "Get Data". Then, select the attributes of interest (harmonized variables `State FIP Code`, `Age`, `Education Number`, `Occupation`, and `Income` in our experiments) and add them to the cart. For your convenience, you can use the direct links at `https://github.com/mosaicrown/mondrian#usa-2018-dataset` (each variable name is a link that redirects to the page at ipums.org that permits to add the variable to the cart). Select the sample of interest (among USA samples, *2018 ACS* in our experiments) and create your data extract. To customize the sample size, set parameter *Persons* (in our experiments *Persons* is set to 510, to obtain a dataset with at least 500,000 tuples). Among the formats available for downloading the dataset, select the csv format and save the downloaded gzip archive in the root folder of the project, with name *usa_<extract_number>.csv.gz*.

Note that the sample of the dataset is randomly extracted at each download from the IPUMS USA web site. Hence, it may differ from the one used in our experimental evaluation.

Tool execution. The procedure to run the experiments has been automated and can be started running command `make artifact_experiments` from the root folder of the project. The procedure operates as follows:

1. it cleans the test environment stopping every Docker container that is still running and removing from HDFS the results produced by the previous runs;
2. it extracts the sample of IPUMS USA dataset to be anonymized from the archive and copies it to the *Spark Driver* volume;
3. it runs the centralized and distributed version of the Mondrian algorithm (see below), and measures the execution time and information loss, storing the results with the following directory structure:

```
mondrian/
 |- percom_artifact_experiments/
 |- |- results/
 |- |- |- runtime_results_<TIMESTAMP>/
 |- |- |- loss_results_<TIMESTAMP>/
```

4. it shuts down all the containers except the *Spark History Server*, which remains available to keep track of the previous runs of the tool.

Centralized version. The centralized version of Mondrian corresponds to the baseline of the experimental results in Section 1.4. The execution of the algorithm can be monitored through the messages shown on the terminal, which report:

1. the schema and the first few tuples of the input dataset;
2. each decision taken by Mondrian to cut the dataset;
3. the schema and the first few tuples of the anonymized dataset;
4. a summary of the information loss measures and the execution time of the algorithm.

The anonymized dataset is in folder *local/anonymized*.

Distributed version. Given a number n of workers available in the distributed system, the tool performs the following steps to execute the distributed version of the Mondrian algorithm:

1. start all the Docker services, initialize HDFS, and submit to the *Spark Driver* our Spark Application;
2. recover the dataset from HDFS and show its structure;
3. retrieve the n-quantiles of the best-scoring attribute of the dataset, showing the score used to decide the optimal cut and the size of the partitions;
4. show the first few tuples of the dataset, complemented with a new attribute containing the id of the quantile to which each tuple belongs and hence the worker to which the tuple is assigned;
5. anonymize the dataset;
6. show the first few tuples of the anonymized dataset, with a summary of the execution time.

The anonymized dataset is in folder *distributed/anonymized*.

1.4 Experimental Results

To assess the scalability of our approach and its limited impact on information loss, we have tested it over a sample of the IPUMS USA dataset [RFG+20], which has become a de-facto benchmark for anonymization solutions. The extracted sample includes 500,000 tuples. We assume the quasi-identifier to include attributes `State FIP Code`, `Age`, `Education Number`, `Occupation`, and the sensitive attribute to be `Income`. We have simulated a distributed environment using a single server through Docker containers. Each node in the architecture in Figure 1.2 runs in a different Docker container. The server is a 12 cores (24 threads) AMD Ryzen 3900X CPU, with 64 GB RAM and 2 TB SSD, running Ubuntu 20.04 LTS, Apache Spark 3.0.1, Hadoop 3.2.1, and Pandas 1.1.3. The distributed algorithm operates over workers equipped with 2GB of RAM and 1 CPU core each. The centralized algorithm relies on 1 CPU core only, with no limitation on the use of the RAM. Our experiments aim at comparing *1)* the execution time and *2)* the information loss of our distributed approach with those of the centralized version of Mondrian.

Execution time. This experiment measures the execution time when computing a 3-anonymous and 2-diverse version of a sample of our sample of the IPUMS USA dataset. The results of the experiments are stored in folder `runtime_results_<TIMESTAMP>`. First, the tool runs the centralized version of the Mondrian algorithm. The results are saved in file *centralized_results.csv*. Then, the tool runs the distributed (Spark-based) version of the Mondrian algorithm, varying the number of workers from 2 to 20. The results are saved in file *spark_based_results.csv*. Besides generating the .csv files with the execution time of the centralized and distributed versions of the algorithm, the tool plots these results generating file *comparison.pdf*.

Figure 1.3 illustrates the execution time (in seconds) for computing a 3-anonymous and 2-diverse version of the IPUMS USA dataset. The figure shows the execution time of our distributed (Spark-based) Mondrian varying the number of workers between 2 and 20. The execution time

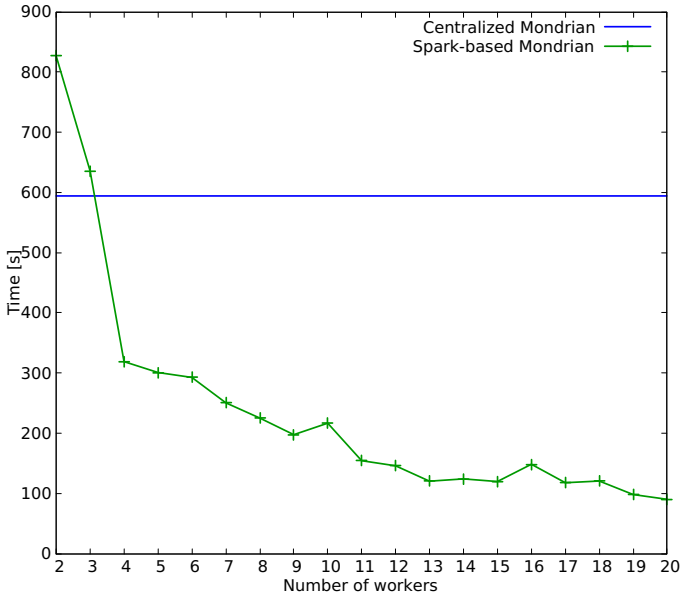


Figure 1.3: Execution times of the centralized version and distributed version varying the number of workers

	100% (centralized)	**0.01% sampling**		
		5 workers	**10 workers**	**20 workers**
DP	1.24*e*7	1.23*e*7 (±4*e*5)	1.26*e*7 (±4*e*5)	1.33*e*7 (±1*e*5)
GCP	6.44	6.47 (±0.08)	6.49 (±0.07)	6.46 (±0.10)

Figure 1.4: DP and GCP information loss with 100% and 0.01% sampling

of the distributed Mondrian decreases, as expected, when the number of workers grows with a saving with respect to the execution time of the centralized Mondrian that ranges from 46% to 85% when using more than 3 workers. This confirms the scalability of our distributed approach. It is interesting to note that the centralized Mondrian is more efficient than the distributed one when the number of workers is low (2 or 3 in our experiments). This is due to the constant initialization time paid by the distributed implementation for setting distribution and interoperation among workers, and by the different libraries used by the centralized implementation (NumPy) and by the distributed implementation (Spark APIs).

Note that the absolute times obtained running our tool may slightly differ from the ones in Figure 1.3, due to the differences in the hardware of the machine used. We however expect the shape of the curves to be similar, proving the scalability of our distributed version of the algorithm.

Information loss. This experiment measures the information loss when computing a 5-anonymous and 2-diverse version of a sample of the IPUMS USA dataset. The results of the experiments are stored in folder `loss_results_<TIMESTAMP>`. The tool first runs the centralized version of the Mondrian algorithm, storing the results in file *centralized_results.csv*. Then, it runs five times the distributed version (with 5, 10, and 20 workers), using a sample including 0.01% of the dataset to determine the most suitable attribute and compute the n-quantiles (with $n = 5$, $n = 10$, and $n = 20$, respectively) for partitioning the dataset among the workers. The results obtained from five runs of the distributed version of the algorithm are stored in file *spark_based_results.csv*. The

tool also generates file *loss_table.csv*, which reports the average and the variance (in the form $\mu \pm \sigma$) of the results in file *spark_based_results.csv*.

We first observe that the information loss caused by distribution can be impacted by: *1)* the number of workers (and hence of partitions), and *2)* the size of the sample used to partition the dataset. Figure 1.4 illustrates the average information loss (and its variance) obtained in five runs of the centralized and distributed (with 5, 10, and 20 workers) Mondrian for computing a 5-anonymous and 2-diverse version of our sample of the IPUMS USA dataset, assuming 0.01% and 100% sampling. In the table, 100% sampling corresponds to the centralized Mondrian, since in our experiments the information loss is substantially not affected by distribution.

The results we obtained confirm that, as expected, information loss grows with the number of workers (i.e., values in DP and GCP lines in Figure 1.4 grow when moving from left to right), but the impact is negligible. Also, the results show that sampling has a very limited impact on information loss (i.e., values obtained with 0.01% sampling are slightly higher than the values obtained with 100% sampling). For instance, GCP increases of less than 2% when passing from the centralized version with 100% sampling to the distributed version with 20 workers and 0.01% sampling. DP has a similar trend.

Note that, since the sample of IPUMS USA dataset is randomly extracted at each download, it may be different from the one used in our experiments and consequently the results might be slightly different from the ones in Figure 1.4. However, we expect the results to have a similar trend, confirming the limited impact of sampling on information loss.

1.5 Conclusions

This chapter illustrated a distributed version of Mondrian that provides scalability without affecting information loss and leveraging an arbitrary number of independent workers. The chapter also describes the tool implementing our distributed Spark anonymization solution. The experimental results confirm that parallelization provides high scalability at a limited cost in terms of information loss. The results obtained by MOSAICrOWN and illustrated in this chapter have been published in [DFF+21a, DFF+21b].

2. Secure Computation of Differentially Private Statistics

In this chapter, we present an efficient secure computation of a differentially private median of the union of two large, confidential data sets.

In distributed private learning, e.g., data analysis, machine learning, and enterprise benchmarking, it is commonplace for two parties with confidential data sets to compute statistics over their combined data. Rank-based statistics (also called order statistics) are values with rank k, i.e., at position k in the sorted data, and encompass min, max, percentiles, inter-quartile ranges, and the median. While we support general rank-based statistics we will focus on the median for illustration purposes. The median is an important statistical method whose value roughly splits the sorted data in half and is used to represent a typical value form the data. We follow related work [AMP10] and define the median to be element at position $\lceil n/2-1 \rceil$ in a data set $D = \{d_0, \ldots, d_{n-1}\}$ of n elements but assume even data size n for better readability in the following. The median is a robust value as few outliers do not change it, unlike the mean. For example, the median of $D = \{1,2,3,1000\}$ is 2, whereas the mean is 251.5 due to skewing by the outlier 1000. Census data usually reports the median income and not the mean income to prevent such skewed results. The *exact* median can be computed securely [AMP10], however, it leaks information about the private data. To protect the data sets, we securely compute a *differentially private* median over the joint data set via the exponential mechanism. The exponential mechanism has a runtime linear in the data universe size and efficiently sampling it is non-trivial. Local differential privacy, where each user shares locally perturbed data with an untrusted server, is often used in private learning but does not provide the same utility as the central model, where noise is only applied once by a trusted server.

Our protocol has a runtime sublinear in the size of the data universe and utility like the central model without a trusted third party. We provide differential privacy for small data sets (sublinear in the size of the data universe) and prune large data sets with a relaxed notion of differential privacy providing limited group privacy. We use dynamic programming with a static, i.e., data-independent, access pattern, achieving low complexity of the secure computation circuit. We provide a comprehensive evaluation over multiple AWS regions (from Ohio to N. Virgina, Canada and Frankfurt) with a large real-world data set with a practical runtime of less than 7 seconds for millions of records.

2.1 Introduction

In distributed private learning two parties A, B, with confidential data sets D_A, D_B respectively, want to compute statistics of their combined data. Example applications are data analysis, machine learning, collaborative forecasting and enterprise benchmarking. The median is an important *robust* statistical method, i.e., a few outliers in the data do not skew the result. The median

is used to represent a "typical" value from a data set and is utilized in enterprise benchmarking, where companies measure their performance against the competition to find opportunities for improvement. Businesses compare, e.g., typical employee salaries per department, bonus payments or sales incentives to better assess their attractiveness for the labor market, and insurance companies use the median life expectancy to adjust insurance premiums. Further, banks compare credit scores of their customers, and financial regulators estimate risks based on loan exposures.

Since the data is sensitive, e.g., salary or health information, the parties want to compute the median without revealing any of their data to each other. A solution to reveal the exact median and nothing else was presented by Aggarwal et al. [AMP10], however, the exact median itself is a value from either D_A or D_B, and, as shown in [DDL78, PP02], median queries can be used to uncover the exact value of targeted individuals. To protect the data sets and hinder targeted inference attacks we also use *differential privacy* [Dwo06, DMNS06]. Inference attacks [DDL78, PP02] rely on median values from the actual data set. The differentially private median, however, is a non-deterministic value from the entire data universe and yet it is close to the actual median with high probability. For small data sets (sublinear in the size of the data universe) we provide differential privacy, and for large data sets we first prune the input using the relaxed notion of differential privacy introduced in [HMFS17]. Instead of considering *neighbors*, i.e., data sets differing in one record, the relaxed notion requires neighbors to also have the same output w.r.t. the initial input pruning. However, we provide empirical evidence that the relaxation is not too restrictive on real-world data sets [Cen17, Kag18, Soo18, ULB18]. A trusted third party, called *curator* in differential privacy literature [DR14], can implement any differentially private algorithms. However, this trusted party requires full access to the unprotected data. To protect the inputs without relying on a trusted third party we use *secure computation* [Gol09], i.e., the parties run a protocol to compute a function on their respective inputs such that nothing about their input is revealed except the function result. In our case, we securely compute the differentially private median via the exponential mechanism, as it provides the best accuracy vs. privacy trade-off for low ε (see our discussion in Section 2.2). The exponential mechanism from McSherry and Talwar [MT07] selects a specific value, like the median, from a data universe $\mathcal{U}$, has a computation complexity linear in the size of the entire data universe [MT07] and efficiently sampling it is non-trivial [DR14]. Also, the exponential mechanism requires exponentiations and divisions, increasing the secure computation complexity. Pettai and Laud [PL15] securely compute the differentially private median using the framework by Nissim et al. [NRS07]. Unlike their work we also considered network delay and used modest hardware[1] and our protocol is still 13 times faster for millions of records with a latency of 25 ms.

We present an efficient protocol to securely compute the differentially private median of the union of two large, confidential data sets with computation complexity sublinear in the size of the data universe. First, the parties prune their own data in a way that maintains their median. Then, they sort and merge the pruned data. The sorted data is used to compute selection probabilities for the entire data universe. Finally, the probabilities are used to select the differentially private median. To optimize the runtime of our protocol we use dynamic programming for the probability computation with a static, i.e., data-independent, access pattern, achieving low complexity of the secure computation circuit. We utilize different cryptographic techniques, garbled circuits as well as secret sharing, to combine their respective advantages, namely, comparisons and arithmetic computations. We simplify the probability and sampling computations to minimize direct access to the data, which reduces secure computation overhead. Furthermore, we compute the required

[1] Our evaluation is performed with AWS t2.medium instances (4 vCPUs, 2GB RAM) compared to the 12-core 3GHZ CPU, 48GB RAM setup of [PL15].

exponentiations for the exponential mechanism without any secure computation.

In summary, the contributions of our protocol combining secure computation and differential privacy are

- selection of the differentially private median of the union of two distributed data sets without revealing anything else about the data,
- an improved runtime complexity sublinear in the size of the data universe achieved by data-independent dynamic programming and input pruning for large data sets,
- a comprehensive evaluation with a large real-world data set with a practical runtime of less than 7 seconds for millions of records even with 100 ms network delay and 100 MBits/s bandwidth.

We note that our protocol can be easily adapted to securely compute the differentially private p^{th}-percentile, i.e., the value larger than $p\%$ of the data. The remainder of this paper is organized as follows: In Section 2.2 we detail the problem description. In Section 2.3 we describe preliminaries for our dynamic programming protocol. In Section 2.4 we explain our approach and introduce definitions. Then, we present our protocol and implementation details for the secure computation of the differentially private median in Section 2.5. We provide a detailed performance evaluation in Section 2.6. We describe related work in Section 2.7 and conclude in Section 2.8.

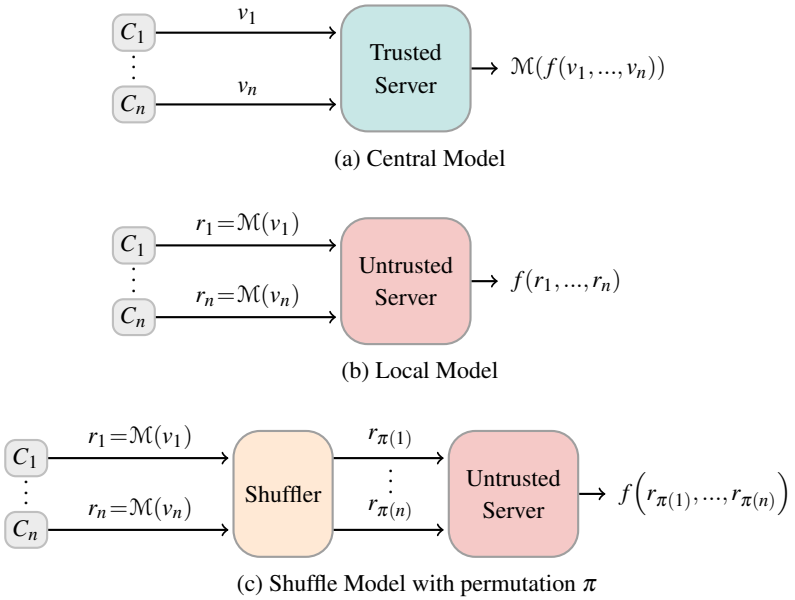
2.2 Problem Description

We consider the problem of two parties computing the differentially private median over their combined data sets. Next, we describe implementation models and basic techniques for differentially private algorithms.

2.2.1 Models for Differentially Private Algorithms

Differentially private algorithms $\mathcal{M}$ can be implemented in different models which are visualized in Figure 2.1. In the *central model* (Figure 2.1a) every client sends their unprotected data to a trusted, central server which runs $\mathcal{M}$ on the clear data. The central model provides the highest accuracy as the randomization inherent to differentially private algorithms, is only applied once. In the *local model* (Figure 2.1b), introduced by [KLN+11], clients apply $\mathcal{M}$ locally and send anonymized values to an untrusted server for aggregation. The accuracy is limited as multiple randomizations occur. It requires enormous amounts of data, compared to the central model, to achieve acceptable accuracy bounds [BEM+17, CSU+19, HKR12, MMP+10]. Specifically, an exponential separation between local and central model for accuracy and sample complexity was shown by [KLN+11]. Recently, an intermediate *shuffle model* (Figure 2.1c) was introduced [BEM+17, CSU+19]: An additional party is added between clients and server in the local model, the shuffler, who does not collude with anyone. The shuffler permutes and forwards the randomized client values. The permutation breaks the mapping between the client and her value, which reduces randomization requirements. The accuracy of the shuffle model lies between the local and central model, however, in general it is strictly weaker than the central model [CSU+19]. As our goal is high accuracy without additional parties, this work dismisses the shuffle model.

To combine the benefits of the local and central model, namely, high accuracy and strong privacy, *secure computation* [Gol09] is used in related work [DKM+06, GX17, RN10, TKZ16].



(a) Central Model

(b) Local Model

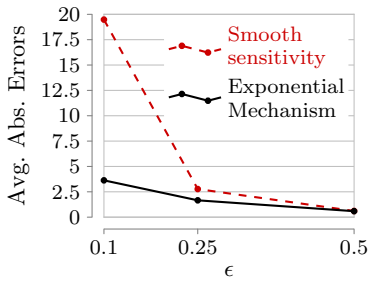
(c) Shuffle Model with permutation π

Figure 2.1: Models for differentially private algorithms $\mathcal{M}$. Client C_i sends a message – raw value v_i or randomized r_i – to a server. The server computes some function f over the messages, and releases the differentially private result.

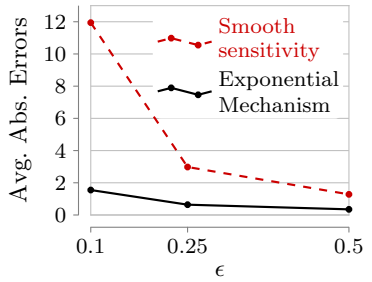
Secure computation allows to simulate central model algorithms in the local model. Secure computation is a cryptographic protocol run between the clients which only reveals the computation's output and nothing more about their sensitive data. Hence, secure computation of the median is superior to distributed computation methods that reveal additional statistics (e.g., histograms or prefix query results) from which to compute a (noisy) median. As Smith et al. [STU17] note, general techniques that combine secure computation and differential privacy suffer from bandwidth and liveness constraints which render them impractical for large data sets. Our contribution is an optimized secure protocol for the differentially private median that runs in seconds on million of records in real-world networks.

2.2.2 Differential Privacy Techniques

Informally, the main techniques to provide differential privacy are *additive noise*, e.g., the Laplace mechanism [DR14], and *probabilistic selection*, namely, the exponential mechanism [MT07]. To compute the differentially private median we use the exponential mechanism [MT07], which provides selection probabilities for possible median values. A simpler, but less accurate, alternative is the *Laplace mechanism* [DR14], which adds noise, sampled from the Laplace distribution, to a function result, i.e., $f(D)+\text{Laplace}(\Delta f/\varepsilon)$. The noise depends on Δf, the *sensitivity* of the function, and a privacy parameter ε formalized later. The sensitivity is the largest difference a single change in *any possible database* can have on the function result. *Smooth sensitivity*, developed by Nissim et al. [NRS07], additionally analyzes the data to provide instance-specific additive noise which is often much smaller. Li et al. [LLSY16] note that the Laplace mechanism is ineffective for the median as the sensitivity, and thus noise, can be high. As mentioned before, the accuracy in the local model is limited [BEM+17, HKR12, MMP+10]. However, even in the central model with smooth sensitivity the exponential mechanism is usually more accurate. To demonstrate this we evaluated the absolute error of the Laplace mechanism with smooth sensitivity and the exponential



(a) Credit Card data [ULB18], first 10^5 payment records in Cents.



(b) Walmart Supply Chain data [Kag18], 175k shipment weights as integers.

Figure 2.2: Absolute errors, averaged for 100 differentially private median computations via exponential mechanism and Laplace mechanism with smooth sensitivity for $\varepsilon \in \{0.1, 0.25, 0.5\}$.

mechanism for real-world data sets [Kag18, ULB18] in Figure 2.2. In general, large differences between elements close to the median or small ε, which corresponds to strong privacy guarantees, increase noise magnitudes and thus errors even with smooth sensitivity. Furthermore, secure computation of smooth sensitivity requires access to the entire dataset or the error further increases[2], which prohibits sublinear secure computation with high accuracy. Thus, our reason for using the exponential mechanism to compute the median is two-fold: It provides the best (known) accuracy for small ε, and, as we will show, it can be implemented as sublinear-time secure computation.

2.3 Preliminaries

Next we introduce preliminaries for differential privacy and secure computation, and some notation.

2.3.1 Notation

We model a database as $D = \{d_0, d_1, \ldots, d_{n-1}\} \in \mathcal{U}^n$. We call $\mathcal{U}$ *data universe* and assume it to be an integer range, i.e., $\mathcal{U} = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$ with $a, b \in \mathbb{Z}$. We note that rational numbers can be expressed as integers via *fixed-point number representation*.[3] To simplify the description we assume the size n of D to be even which can be ensured by padding. Then, the median is the value $d_{n/2-1}$ in sorted D. We denote with $I_D = \{0, \ldots, n-1\}$ the set of indices for D and refer to non-distinct data elements as *duplicates*, i.e., $d_i = d_j$ with $i \neq j$ $(i, j \in I_D)$. We apply union under bag semantics, i.e., $D_A \cup D_B$ is a *bag* containing elements from $\mathcal{U}$ as often as they appear in data sets D_A and D_B combined[4]. We treat the difference of two bags, $D_A \backslash D_B$, as a *set* containing only elements from D_A that are not also in D_B.

[2] Smooth sensitivity approximations exist that provide a factor of 2 approximation in linear-time, or an additive error of $\max(\mathcal{U})/\text{poly}(n)$ in sublinear-time [NRS07, Section 3.1.1]. Note that this error e is w.r.t. smooth sensitivity s and the additive noise is even larger with Laplace $((s+e)/\varepsilon)$.

[3] A binary number of bit-length b can represent $d \in \mathbb{Q}$ as $d' \in \mathbb{Z}$ if $d = d' \cdot 2^{-f}$ with $-2^{b-1}+1 \leq d' \leq 2^{b-1}-1$ and scaling factor 2^{-f}, $f \in \mathbb{N}$.

[4] This interpretation of union is equivalent to the *sum* function for bags.

2.3.2 Differential Privacy

Differential privacy introduced by Dwork et al. [DMNS06] [Dwo06,DMNS06] is a privacy notion, adopted by major technology companies [DKY17, EPK14, Tea17, WWD16]. Differential privacy enables one to learn statistical properties of a data set while protecting the privacy of any individual contained in it. Data sets D, D' are called *neighbors* or *neighboring*, denoted with $D \simeq D'$, when data sets D can be obtained from D' by adding or removing one element, i.e., $D = D' \cup \{x\}$ with $x \in \mathcal{U}$ or $D = D' \backslash \{y\}$ with $y \in D'$.

Informally, a differentially private algorithm limits the impact that the presence or absence of any individual's data in the input database can have on the *distribution* of outputs. The formal definition is as follows:

Definition 1 (Differential Privacy)**.** A mechanism $\mathcal{M}$ satisfies ε-differential privacy, where $\varepsilon \geq 0$, if for all neighboring data sets D and D', and all sets $S \subseteq \text{Range}(\mathcal{M})$

$$\Pr[\mathcal{M}(D) \in S] \leq \exp(\varepsilon) \cdot \Pr[\mathcal{M}(D') \in S],$$

where $\text{Range}(\mathcal{M})$ denotes the set of all possible outputs of mechanism $\mathcal{M}$.

The above definition holds against an unbounded adversary, however, due to our use of cryptography we assume a polynomial-time bounded adversary. Mironov [MPRV09] define indistinguishable computationally differential privacy (IND-CDP) for two-party computation (2PC) with computationally bounded parties. The presented definition is according to [HMFS17] for parties A, B with data sets D_A, D_B, privacy parameters $\varepsilon_A, \varepsilon_B$ and security parameter λ. Furthermore, $\texttt{VIEW}_A^\Pi$ denotes the view of A during the execution of protocol Π.

Definition 2 (IND-CDP-2PC)**.** A two-party protocol Π for computing function f satisfies $(\varepsilon_A(\lambda), \varepsilon_B(\lambda))$-indistinguishable computationally differential privacy (IND-CDP-2PC) if $\texttt{VIEW}_A^\Pi(D_A, \cdot)$ satisfies $\varepsilon_B(\lambda)$-IND-CPA, i.e., for any probabilistic polynomial-time (in λ) adversary $\mathcal{A}$, for any neighboring data sets (D_B, D'_B)

$$\begin{aligned}&\Pr[\mathcal{A}(\texttt{VIEW}_A^\Pi(D_A, D_B)) = 1] \\ \leq &\exp(\varepsilon_B) \cdot \Pr[\mathcal{A}(\texttt{VIEW}_A^\Pi(D_A, D'_B)) = 1] + \text{negl}(\lambda).\end{aligned}$$

Likewise for B's view for any neighbors (D_A, D'_A) and ε_A.

For notational convenience let $\varepsilon = \varepsilon_A = \varepsilon_B$. We operate in the *semi-honest* model [Gol09] (also called honest-but-curious) where participants do not deviate from the protocol but try to extract as much information from the protocol transcript as possible. A protocol is consider secure in the semi-honest model when the transcript does not reveal anything beyond the computed functionality.

2.3.3 f-neighboring

He et al. [HMFS17] introduced the notion of *f-neighbors*: neighbors that also have the same output w.r.t. to a function f. For our security proof we require f-neighboring and adapt it to our scenario.

Definition 3 (f-Neighbor)**.** Given function $f : \mathcal{U}^k \times \mathcal{U}^l \to \mathcal{O}$, $k, l \in \mathbb{N}$, and $D_A \in \mathcal{U}^k$. Data sets D_B and D'_B are f-neighbors w.r.t. $f(D_A, \cdot)$ if

1. they are neighbors, and
2. $f(D_A, D_B) = f(D_A, D'_B)$.

f-neighboring for D_B is similarly defined.

In [HMFS17] f-neighboring is applied to record matching, where neighbors differ in at most one *non-matching* record. In our scenario f is input pruning, the first step of our protocol which reduces the input set size and we denote it as PRUNE. PRUNE is a partial execution of comparison-based pruning from [AMP10] described in Section 2.4.4. We distinguish two forms of pruning: deterministic and randomized. *Deterministic* pruning, such as PRUNE, might differ between neighboring data sets and thus potentially violate differentially privacy for its common neighboring notion. By considering PRUNE-neighbors, where pruning outputs are the same, neighboring data sets cannot be distinguished, and differential privacy holds. To verify that PRUNE-neighboring is not too restrictive and can be used in real-world applications we evaluated neighboring data sets from real-world data sets [Cen17, Kag18, Soo18, ULB18] and found they are all also PRUNE-neighboring albeit with limited group privacy. See Section 2.6 for details of the experiment. In *randomized* pruning each comparison result is randomized. The probability that the half of the data containing the median is never discarded decreases exponentially in the number of comparisons [HLM17]. Hence, accuracy is significantly impacted with high probability and we dismiss randomized pruning in favor of PRUNE-neighboring.

2.3.4 Exponential Mechanism

The exponential mechanism, introduced by McSherry and Talwar [MT07], expands the application of differential privacy to functions with non-numerical output, and when the output is not robust to additive noise. The exponential mechanism selects a result from a fixed set of outputs $\mathcal{O}$ while satisfying differential privacy. The mechanism is exponentially more likely to select "good" results where "good" is quantified via a utility function $u(D,o)$ which takes as input a data set $D \in \mathcal{U}^n$ and a potential output $o \in \mathcal{O}$. The utility function provides a utility score for o w.r.t. D and all possible output values from $\mathcal{O}$. Informally, a higher score means the output is more desirable and its selection probability is increased accordingly. The formal definition is according to [LLSY16].

Definition 4 (Exponential Mechanism)**.** For any utility function $u : (\mathcal{U}^n \times \mathcal{O}) \to \mathbb{R}$ and a privacy parameter ε, the exponential mechanism $\mathcal{M}_u^\varepsilon(D)$ outputs $o \in \mathcal{O}$ with probability proportional to $\exp\left(\frac{\varepsilon u(D,o)}{2\Delta u}\right)$, where

$$\Delta u = \max_{\forall o \in \mathcal{O}, D \simeq D'} \left| u(D,o) - u\left(D',o\right) \right|$$

is the sensitivity of the utility function. That is,

$$\Pr[\mathcal{M}_u^\varepsilon(D) = o] = \frac{\exp\left(\frac{\varepsilon u(D,o)}{2\Delta u}\right)}{\sum_{o' \in \mathcal{O}} \exp\left(\frac{\varepsilon u(D,o')}{2\Delta u}\right)}. \tag{2.1}$$

Median Utility Function

We focus on the median and use the median utility function from Li et al. [LLSY16, Section 2.4.3] where $\text{rank}_D(x)$ denotes the number of elements in D smaller than x.

Definition 5 (Median utility function)**.** The median utility function $u_{\text{med}} : (\mathcal{U}^n \times \mathcal{U}) \to \mathbb{Z}$ gives a utility score for each $x \in \mathcal{U}$ w.r.t. $D \in \mathcal{U}^n$ as

$$u_{\text{med}}(D,x) = - \min_{\text{rank}_D(x) \le j \le \text{rank}_D(x+1)} \left| j - \frac{n}{2} \right|.$$

Note that for the median $\mathcal{O} = \mathcal{U}$, i.e., every universe element has to be considered as a potential output. The sensitivity of u_{med} is $1/2$ since adding an element increases $n/2$ by $1/2$ and j either increases by 1 or remains the same [LLSY16]. Thus, the denominator $2\Delta u$ in the exponents of Equation (2.1) equals 1, and we will omit it in the rest of this work. The intuition behind this utility definition is to use the rank of elements to quantify their "closeness" to the median. The median itself has the highest utility value, 0, all other elements have negative utility. The further away an element in a sorted data set (i.e., its rank) is from the median position, the smaller its utility. Note that Definition 5 can be adapted to select elements of arbitrary rank k, e.g., to find the 25^{th}- and 75^{th}-percentile. In this work we focus on the secure computation of the differentially private median but this can easily be extended to securely compute the differentially private k^{th}-ranked element.

2.3.5 Secure Computation

We use secret sharing as well as garbled circuits as addition and scalar multiplication are more efficient with the former whereas comparisons can be more efficiently implemented as boolean circuits with the latter.

Additive Secret Sharing

We require all values to be in the ring $\mathbb{Z}_{2^b}$ and perform all operations modulo $\mathbb{Z}_{2^b}$. In additive p-out-of-p secret sharing a party P_i, $1 \le i \le p$, "splits" its secret value $s \in \mathbb{Z}_{2^b}$ into p shares and all shares are required to reconstruct the secret. First, P_i creates uniformly random values $s_1, \ldots, s_{p-1} \in \mathbb{Z}_{2^b}$. Then, P_i sets $s_p = s - \sum_{i=1}^{p-1} s_i$. Intuitively, a shared secret is reconstructed by adding all shares together, i.e., $s = \sum_{i=1}^{p} s_i$. Privacy follows from the fact that shares $s_1, \ldots, s_p$ are uniformly random and the sum of any strict subset of the shares is also random. We denote the sharing of s as $\langle s \rangle = (s_1, \ldots, s_p)$. Addition with shared secret values $\langle s \rangle, \langle r \rangle$ is straightforward since $\langle s \rangle + \langle r \rangle = (s_1 + r_1, \ldots, s_p + r_p)$, as is multiplication with a public value $t \in \mathbb{Z}_{2^b}$ where $t\langle s \rangle = (ts_1, \ldots, ts_p)$. We also write $\langle s \rangle_{P_j}$ instead of s_j to highlight that it is P_j's share of s.

In our implementation we use the ring $\mathbb{Z}_{2^{64}}$ as computations modulo 2^{64} are commonly supported on standard CPUs.

Garbled Circuits

A garbled circuit, first described by Yao [Yao86], is a general technique to securely evaluate any function by implementing it as a boolean circuit and "garbling" each gate's truth table. Informally speaking, given two parties, the four possible inputs of a garbled table are not plaintext bits but random labels. One party is the *garbler* who garbles the gates and creates the labels. The other party, called *evaluator*, receives the garbled circuit and evaluates it. The garbler includes all her input labels in the garbled circuit (which look random to the evaluator). However, the garbler cannot learn the evaluator's input and cannot send both input labels per gate to the evaluator, otherwise the garbler's input will be revealed. To solve this problem 1-out-of-2 *oblivious transfer* (OT) [EGL85, Rab81] is used: The evaluator receives only her input label and the garbler remains

oblivious. Given the input labels for both parties the evaluator can determine (decrypt) the output label for a gate and use it as input for the next gate. An output translation table, also provided by the garbler, maps the final random output label to the plain result.

Bellare et al. [BHR12] formalize a *garbling scheme* as the tuple of algorithms $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$, where Gb is probabilistic and all others are deterministic. A *string* is defined as a sequence of bits of finite length.

- $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, f)$: Takes as input a security parameter $\lambda \in \mathbb{N}$ and the string f describing the original function to evaluate, $\mathsf{ev}(f, \cdot)$, and outputs string F describing the *garbled function*, $\mathsf{Ev}(F, \cdot)$, string e describing an *encoding function*, $\mathsf{En}(e, \cdot)$, and string d describing a *decoding function*, $\mathsf{De}(d, \cdot)$, as defined in the following.
- $X \leftarrow \mathsf{En}(e, x)$ is an *encoding function*, described by string e, that maps an *initial input* $x \in \{0,1\}^n$ to a *garbled input* X.
- $y \leftarrow \mathsf{De}(d, Y)$ is a *decoding function*, described by string d, that maps a *garbled output* Y to a *final output* y.
- $Y \leftarrow \mathsf{Ev}(F, X)$ is an *evaluation function*, described by string F, that maps a garbled input X to a *garbled output* Y.
- $y \leftarrow \mathsf{ev}(f, x)$ is an *evaluation function*, described by string f, that maps the input x to the output y, where $\mathsf{ev}(f, \cdot) : \{0,1\}^n \rightarrow \{0,1\}^m$ is the *original function* we want to garble, and $n = f.n, m = f.m$ depend on f and must be computable from it in linear-time.

2.4 Building Blocks for DP Median Selection

We implement an efficient, secure computation of the exponential mechanism which selects the differentially private median from the entire data universe $\mathcal{U}$. There are two challenges for secure computation of the exponential mechanism:

- the runtime complexity is linear in $|\mathcal{U}|$ as probabilities for *all* possible outputs in $\mathcal{U}$ are computed,
- the general exponential mechanism is too inefficient with general secure computation as it requires $|\mathcal{U}|$ exponentiations and divisions.

In this section we present building blocks for our practically efficient, sub-linear time protocol overcoming those challenges.

2.4.1 Overview

For now we focus on a single data set as we later prune and merge the data sets from the two parties into one data set. For data set D with universe $\mathcal{U}$ we compute the median selection probabilities for all of $\mathcal{U}$ using only D by utilizing dynamic programming. To compute the probabilities efficiently we first define a simplified utility function *utility*, which computes utility for all universe elements but only requires D as input, in Section 2.4.2. The simplified *utility* provides incorrect utility scores in the presence of duplicates. Thus, we define *gap* to discard these incorrect scores and compute the median selection probabilities, denoted as *weight*. The sum of these probabilities

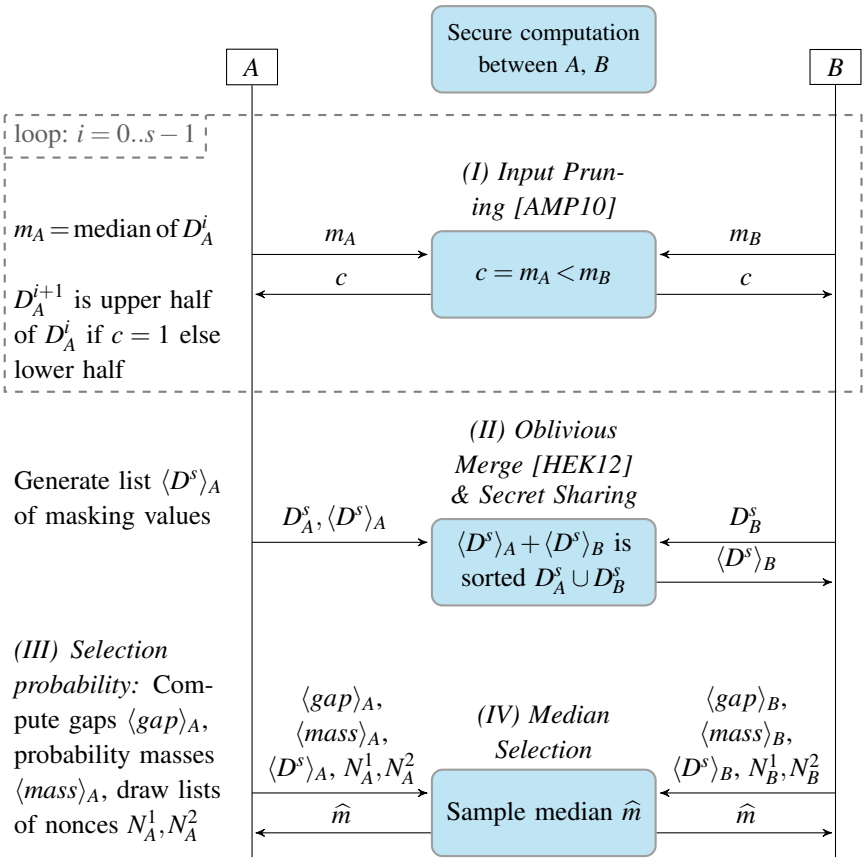


Figure 2.3: High-level protocol overview with comments for A, where s is the number of pruning steps, D_A^0 is sorted D_A, and $\langle D^s \rangle_A, \langle gap \rangle_A, \langle mass \rangle_A$ are A's shares for all values d_i^s, gaps $gap(i)$, and masses $mass(i)$ respectively ($i \in \{0, \ldots, |D^s| - 1\}$).

is the basis for the cumulative distribution function, which we denote with *mass*. Then, we sample the differentially private median based on *mass* with inverse transform sampling described in Section 2.4.3. To further reduce secure computation complexity we prune the input D in Section 2.4.4. A high-level overview of our protocol is visualized in Figure 2.3, and we present our full protocol in Section 2.5. In the first step, the parties prune their input. Then, they securely merge and secret share their pruned data. In the third step they compute selection probabilities and, in the last step, sample the differentially private median.

Note that in the following we define *gap*, *utility*, and *weight* such that direct access to the data D – and therefore the need for secure computation – is minimized: Each party can compute *utility* and *weight* without any access to D. Furthermore, *gap* has a static access pattern in dynamic programming, independent of the elements in (sorted) D, which makes the *gap* function *data-oblivious*, i.e., an attacker who sees the access pattern cannot learn anything about the sensitive data.

2.4.2 Utility with Static Access Pattern

Recall that the exponential mechanism evaluates the utility function u_{med} for *all* elements in the data universe $\mathcal{U}$. However, per definition of u_{med} certain outputs have the same utility, namely, duplicates and elements in $\mathcal{U} \setminus D$. We use this observation to simplify the median utility definition and evaluate it only for elements in D instead of the entire universe $\mathcal{U}$.

Definition 6 (Median utility function)**.** Let data set $D \in \mathcal{U}^n$ be sorted. The *median utility function* $utility : I_D \to \mathbb{Z}$ scores the utility of an element of D at position $i \in I_D$ as

$$utility(i) = \begin{cases} i - \frac{n}{2} + 1 & \text{if } i < \frac{n}{2} \\ \frac{n}{2} - i & \text{else} \end{cases}.$$

First, we prove the equivalence of utility function $utility$ and u_{med} only for distinct data ($D \subseteq \mathcal{U}$) then we define gap to help with the utility computation for data sets with duplicates.

Theorem 1 (Utility equivalence)**.** For $D \subseteq \mathcal{U}$ and index $i \in I_D$ we have

$$u_{\text{med}}(D,x) = utility(i)$$

for $x \in [d_i, d_{i+1})$ with $i < n/2$ and $x \in (d_{i-1}, d_i]$ with $i \geq n/2$.

Proof. First, we show that all elements in $x \in [d_i, d_{i+1})$ for $i < n/2$ and $x \in (d_{i-1}, d_i]$ for $i \geq n/2$ have the same utility. The utility u_{med} of an element $x \in \mathcal{U}$ is based on a rank from the set $S_x = \{j \mid \text{rank}_D(x) \leq j \leq \text{rank}_D(x+1)\}$ according to Definition 5. For $i < n/2$, $x \geq d_i$ and $x+1 < d_{i+1}$ we have $\text{rank}_D(x+1) = \text{rank}_D(d_{i+1})$. All elements in the open range (d_i, d_{i+1}) have the same rank set $S = \{\text{rank}_D(x+1)\}$. The rank set for d_i, S_{d_i}, is a superset of S that also includes ranks smaller than $\text{rank}_D(x+1)$. However, $\text{rank}_D(x+1) = S_{d_i} \cap S$ minimizes the term $|\text{rank}_D(x+1) - n/2|$ since it is the value closest to $n/2$. Thus, all elements in the half-open range $[d_i, d_{i+1})$ have the same utility. Analogously, for $i \geq n/2$ elements in $(d_{i-1}, d_i]$ have the same utility.

For $i \in I_D$ and sorted $D \subseteq \mathcal{U}$ we have $\text{rank}_D(d_i) = i$ and $S_{d_i} = \{\text{rank}_D(d_i), \text{rank}_D(d_i+1)\} = \{i, i+1\}$. Thus,

$$\begin{aligned} u_{\text{med}}(D,d_i) &= -\min_{j \in \{i,i+1\}} \left| j - \frac{n}{2} \right| \\ &= \begin{cases} i + 1 - \frac{n}{2} & \text{if } i < \frac{n}{2} \\ \frac{n}{2} - i & \text{else} \end{cases} \\ &= utility(i). \end{aligned}$$

□

Thus, the sensitivity of $utility$ is the same as u_{med}. We stress that $utility(i)$ only depends on the *position* i in the sorted data. In essence, we assume all elements in D are distinct, in this case $utility(i) = u_{\text{med}}(D, d_i)$.

Each party can compute $utility$ (Definition 6) without any access to D, and gap (Definition 7) has a static access pattern, independent of the elements in (sorted) D, which makes the gap function *data-oblivious*, i.e., an attacker who sees the access pattern cannot learn anything about D. Figure 2.4 visualizes how we compute $utility$ and gap with static access pattern over sorted data D.

To only retain the correct utility in the presence of duplicates we define gap next.

Definition 7 (Gap)**.** The *gap function* $gap : I_D \to \mathbb{N}_0$ provides the number of consecutive elements in $\mathcal{U}$ with the same utility as d_i with

$$gap(i) = \begin{cases} d_{i+1} - d_i & \text{if } i < \frac{n}{2} - 1 \\ 1 & \text{if } i = \frac{n}{2} - 1 \\ d_i - d_{i-1} & \text{else} \end{cases}. \tag{2.2}$$

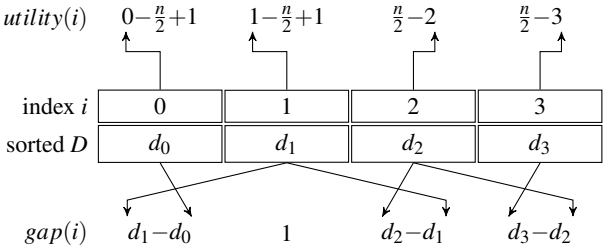


Figure 2.4: *utility* and *gap* computed on sorted D with static access pattern.

Note that *gap* is defined for all n indices although there are only $n-1$ gaps between values in D. We set the median's gap to 1 as it is the only element not contained in the union of all half-open ranges. If D contains duplicates *gap* is zero for all except the duplicate closest to the median. Thus, a gap value of zero indicates incorrect utility for a duplicate and we use this to eliminate such utility values in the following.

First, with the help of *utility* we define the unnormalized selection probability, which we call *weight*.

Definition 8 (Weight)**.** The *weight function* $weight : I_D \to \mathbb{R}$ gives the unnormalized selection probability for an element at index $i \in I_D$ as

$$weight(i) = \exp(\varepsilon \cdot utility(i))$$

where ε is the privacy parameter from Definition 1.

Then, we use *weight* and *gap* to define the probability mass of elements with the same utility, which we call *mass*.

Definition 9 (Mass)**.** The *probability mass function* $mass : I_D \to \mathbb{R}$ at $i \in I_D$ is

$$mass(i) = \sum_{h=0}^{i} weight(h) \cdot gap(h).$$

To ensure that *mass* covers *all* elements in $\mathcal{U}$ we append the smallest and largest universe element to the beginning resp. end of D before computing *mass*. Now, we show that *mass* is the (unnormalized) cumulative density function for the distribution defined by $\mathcal{M}_u^\varepsilon(D)$.

Theorem 2. Let $\mathcal{O} = \{d_0, \ldots, d_i\} \subseteq \mathcal{U}$ with D sorted, $\min(\mathcal{U}), \max(\mathcal{U}) \in D$ and $i \in I_D$, then

$$\frac{mass(i)}{R} = \sum_{o \in \mathcal{O}} \Pr[\mathcal{M}_u^\varepsilon(D) = o],$$

with $u = u_{\text{med}}$ and normalization $R = \sum_{o' \in \mathcal{U}} \Pr[\mathcal{M}_u^\varepsilon(D) = o']$.

Proof. Without duplicates $utility = u_{\text{med}}$ (Theorem 1), thus, $weight(i) = \exp(\varepsilon \cdot u_{\text{med}}(D, d_i))$ for $i \in I_D$. With duplicates *weight* can produce incorrect values, however, $weight(i) \cdot gap(i) = 0$ as *gap* is zero for all duplicates except the one closest to the median. In other words, we eliminate weights based on incorrect utility values as they do not alter the sum $mass[i] = \sum_{h=0}^{i} weight(h) \cdot gap(h)$.

On the other hand, $gap > 0$ indicates the number of consecutive elements in $\mathcal{U}$ with same utility, and $weight(i) \cdot gap(i)$ is their unnormalized probability mass. Thus, $mass[i]$ equals the sum of unnormalized probabilities for elements in $\mathcal{O} = \{\min(\mathcal{U}), \ldots, d_i\}$, and $mass[i]/R$ equals normalized probabilities $\sum_{o \in \mathcal{O}} \Pr[\mathcal{M}_u^\varepsilon(D) = o]$. □

Table 2.1: u_{med} compared with *utility* with static access pattern and *gap* for sorted $D = \{2,2,6,6,7,7\}$, $\mathcal{U} = \{1,\ldots,10\}$. To cover utility for all of $\mathcal{U}$ we add $\min(\mathcal{U}), \max(\mathcal{U})$ to D.

index i	0	1	2	–	3	4	5	6	–	7
sorted D	1	2	2	3,4,5	6	6	7	7	8,9	10
$\text{rank}_D(\cdot)$	0	1	1	3	3	3	5	5	7	7
$u_{\text{med}}(D,\cdot)$	-3	-1	-1	-1	0	0	-1	-1	-3	-3
$utility(i)$	-3	-2	-1	-1	0	0	-1	-2	-3	-3
$gap(i)$	1	0	4	–	1	0	1	0	–	3

$\min(\mathcal{U})$, $\max(\mathcal{U})$ Missing elements $\mathcal{U}\backslash D$

An example for *utility* and *gap* can be found in Table 2.1. It illustrates that *utility* for sorted D is just a sequence that first increases, then decreases after the median. As mentioned above, we add $\min(\mathcal{U})$ to the beginning and $\max(\mathcal{U})$ to the end of D (highlighted in light gray in Table 2.1). The utility for "missing elements" in $\mathcal{U}\backslash D$ (dark gray columns) is the same as for the preceding or succeeding element in D. Furthermore, *gap* is zero for the duplicates furthest away from the median and otherwise indicates the number of consecutive elements in $\mathcal{U}$ with the same utility (e.g., $gap(2) = 4$ as $2,3,4,5$ have the same utility as $d_2 = 2$).

2.4.3 Median Sampling

We use *inverse transform sampling* to sample the differentially private median from the cumulative distribution function *mass* by finding an index $j \in I_D$[5] such that $mass(j-1) \leq r < mass(j)$ for a uniform random r. Inverse transform sampling simulates any distribution via the uniform distribution and the intuition behind it is best illustrated with a toy example: Given two elements a, b with selection probabilities $0.7, 0.3$, respectively, we can fill an array A of, say, size 100 with 70 copies of a and 30 copies of b. Next, we choose an uniform random index r of A and output the element at that position, i.e., $A[r]$. Thus, we output a with probability 0.7 and b with probability 0.3 albeit r was drawn at uniform random.

After the sampling step, we select an element at uniform random among the $gap(j)$ consecutive elements with the same utility as the element at index j. Now, with our simplified utility, we do not need to iterate over all elements in $\mathcal{U}$, but only over elements in D while still covering all "missing" elements ($\mathcal{U}\backslash D$) via *gap*.

2.4.4 Input Pruning with non-decreasing Utility

However, n might be large and we show how to prune D via [AMP10] before applying our median selection. Next, we explain pruning, define accuracy, and present the maximum pruning steps for a given accuracy.

PRUNE is a technique used by Aggarwal et al. [AMP10] to securely find the median of two parties A, B with respective data sets D_A, D_B. We assume the data size of each party, i.e., $|D_A|, |D_B|$, to be known, however, it can be hidden via additional padding. As preprocessing, the parties A, B sort their respective data sets D_A, D_B and only retain the smallest $k = \lceil |D_A| + |D_B| \rceil / 2$ values[6]. Then, they pad the remaining data with $-\infty, +\infty$ to be of size $2^{\lceil \log_2(k) \rceil}$ in a way that preserves the

[5] For notational convenience let $j - 1 < 0$ be 0.

[6] If the data contains duplicates, $\lceil \log_2 n \rceil + 1$ bits are added to the element's binary representation to make it unique, which is required for the security proof from [AMP10, Section 3.2]. We implement the uniqueness encoding but omit it in the presented protocol to simplify its description.

position of the median (see Appendix 2.5.1 for details). In each pruning step the parties compute their respective medians, m_A, m_B, perform a secure comparison $m_A < m_B$, and use the result to discard the halves of their data that cannot contain their mutual median, i.e., A retains the upper half of D_A if $m_A < m_B$ and the lower half otherwise, B does the opposite. After $\log n$ iterations only their exact mutual median remains. We denote data sets D_A, D_B after pruning step s as D_A^s, D_B^s and their union as D^s. Note that PRUNE does not violate differential privacy as we only consider PRUNE-neighboring data sets with the same comparison results similar to [HMFS17]. The median m of D is also the median of D^s as shown in [AMP10, Lemma 1]. How the data D is distributed among parties changes the intermediary outcome of the pruning, i.e., what elements remain in D_A^s, D_B^s. However, utility depends on an element's closeness to the median which remains or increases if elements in between are removed as we show next.

Theorem 3. The input pruning from [AMP10] does not decrease utility.

Proof. Let $D_A = \{a_1, \ldots, a_m\}, D_B = \{b_1, \ldots, b_m\}$ with $a_1 < a_2 < \cdots < a_m$ and $b_1 < b_2 < \cdots < b_m$ (otherwise we use padding and uniqueness encoding from [AMP10]). Let $a_i^s = D_A^s[i]$, i.e., the element at index i in the data of A after pruning step s. If some indices i, j, k exist such that $a_i^{s-1} < b_j^{s-1} \leq b_k^{s-1} < a_{i+1}^{s-1}$ where $b_j^{s-1}, \ldots, b_k^{s-1}$ are not in D_B^s but a_i^{s-1} is in D_A^s then pruning step s removed $b_j^{s-1}, \ldots, b_k^{s-1}$ but neither a_i^{s-1} nor a_{i+1}^{s-1}, one of which is further away from the median than $b_j^{s-1}, \ldots, b_k^{s-1}$. However, the utility of such a removed element either remains the same (it is a duplicate of a remaining element), or increases, i.e., they have the utility of their predecessor (resp., successor) in D^s. Since one of the elements a_i^{s-1}, a_{i+1}^{s-1} is closer to the median after pruning step s than before, its utility increases and so does the utility for all elements between a_i^{s-1} and a_{i+1}^{s-1}.

If no such indices i, j, k exist, then we only remove the elements furthest away from the median and the utility for remaining elements is unchanged. The utility for removed element x either remains the same (x is equal to a remaining element) or increases. The latter is due to the fact that removed elements have the same rank-based distance to the median, either $\text{rank}_{D^s}(x) = 0$ or $\text{rank}_{D^s}(x) = |D^s|$. Since $|D^s| < |D^{s-1}|$ we have $u_{\text{med}}(D^s, x) > u_{\text{med}}(D^{s-1}, x)$. □

An example of non-decreasing utility after pruning is shown in Table 2.2 for unique elements. For example, element a_1 has utility -3 before pruning, after pruning its utility increases to -2, whereas the utility for b_2, a_3 remain as before.

Table 2.2: Utility does not decrease for sorted $D = D_A \cup D_B$ before and after one pruning step with $D_A = \{a_1, \ldots, a_4\}$, $D_B = \{b_1, \ldots, b_4\}$.

D	a_1	b_1	a_2	b_2	a_3	b_3	a_4	b_4
$u_{\text{med}}(D, \cdot)$	-3	-2	-1	0	0	-1	-2	-3
D^1	–	b_1	–	b_2	a_3	–	a_4	–
$u_{\text{med}}(D^1, \cdot)$	-2	-1	-1	0	0	-1	-1	-2

Before we can find the maximum number of pruning steps we first define what accuracy we want to maintain after pruning. We separate the universe $\mathcal{U}$ in two disjunct sets of *remaining elements* $\mathcal{R}$ and *pruned elements* $\mathcal{P}$ where $\mathcal{R} = \{x \in \mathcal{U} \mid \min(D^s) \leq x \leq \max(D^s)\} \subseteq \mathcal{U}$ and $\mathcal{P} = \{x \in \mathcal{U} \mid x < \min(D^s) \text{ or } x > \max(D^s)\} = \mathcal{U} \backslash \mathcal{R}$. Note that $\mathcal{R}$ contains the universe elements closest to the median.

Definition 10 (Accuracy)**.** Let $u = u_{\text{med}}$, then *accuracy* is

$$p_{\mathcal{R}} = 1 - p_{\mathcal{P}} = \sum_{x \in \mathcal{R}} \Pr[\mathcal{M}_u^{\varepsilon}(D^s) = x],$$

i.e., $p_{\mathcal{R}}$ is the probability mass of all remaining elements.

With accuracy $p_{\mathcal{R}} > 0.5$ it is more likely to select the differentially private median among $\mathcal{R}$ than among $\mathcal{P}$. In our evaluation we use accuracy $p_{\mathcal{R}} = 0.9999$. The number of pruning steps s enables a trade-off between accuracy $p_{\mathcal{R}}$ and computation complexity: smaller s leads to higher accuracy and larger s translates into smaller input size for the secure computation. We are interested in the maximum number of pruning steps such that it is more likely to select an element from $\mathcal{R}$ instead of $\mathcal{P}$.

Theorem 4 (Upper Bound for Pruning Steps)**.** Let D be a data set with data universe $\mathcal{U}$, $\varepsilon > 0$, and $0 < \alpha < 1$. The upper bound for pruning steps s fulfilling $p_{\mathcal{R}} \geq \alpha$ is

$$\lfloor \log_2(\varepsilon n) - \log_2\left(\log_e\left(\frac{\alpha}{1-\alpha}(|\mathcal{U}|-1)\right)\right) - 1 \rfloor.$$

Proof. We find the maximum number of pruning steps s by examining what the maximum probability mass $p_{\mathcal{P}}$ for pruned elements can be.

First, note that the utility for all $x \in \mathcal{P}$ is the same independent of the values in D^s: Half of the values in $\mathcal{P}$ are smaller (resp., larger) than the median m of D^s, i.e., $\text{rank}_{D^s}(x) = 0$ if $x < m$ and $\text{rank}_{D^s}(x) = |D^s|$ otherwise. Thus, $u_{\text{med}}(D^s, x) = -\left|0 - \frac{|D^s|}{2}\right| = -\left||D^s| - \frac{|D^s|}{2}\right| = -\frac{n}{2^{s+1}}$ since $|D^s| = \frac{n}{2^s}$. (Recall that D is padded before pruning such that n is a power of two.)

As the utility, and thus selection probability, is the same for all elements in $\mathcal{P}$ the probability mass $p_{\mathcal{P}}$ is maximized if $|\mathcal{P}|$ is maximized. The maximum for $|\mathcal{P}|$ is $|\mathcal{U}| - 1$ as $\mathcal{R}$ must contain at least one element, the median m.

Let $p'_{\mathcal{R}}, p'_{\mathcal{P}}$ be the unnormalized probability masses $p_{\mathcal{R}}, p_{\mathcal{P}}$ respectively, then

$$p'_{\mathcal{R}} = \exp(\varepsilon u_{\text{med}}(D^s, m)) = 1$$

since $\mathcal{R} = \{m\}$ and $u_{\text{med}}(D^s, m) = 0$, and

$$p'_{\mathcal{P}} = (|\mathcal{U}| - 1)\exp\left(-\varepsilon \frac{n}{2^{s+1}}\right)$$

with normalization term $R = p'_{\mathcal{P}} + p'_{\mathcal{R}}$. Now accuracy $p_{\mathcal{R}}$ of at least α is equivalent to

$$\begin{aligned}
\alpha \leq \frac{p'_{\mathcal{R}}}{R} &= \frac{1}{(|\mathcal{U}|-1)\exp\left(-\frac{\varepsilon n}{2^{s+1}}\right) + 1} \\
\Leftrightarrow \exp\left(-\frac{\varepsilon n}{2^{s+1}}\right) &\leq \frac{1-\alpha}{\alpha(|\mathcal{U}|-1)} \\
\Leftrightarrow \log_e\left(\frac{\alpha(|\mathcal{U}|-1)}{1-\alpha}\right) &\leq \frac{\varepsilon n}{2^{s+1}} \\
\Leftrightarrow s &\leq \log_2\left(\frac{\varepsilon n}{\log_e\left(\frac{\alpha}{1-\alpha}(|\mathcal{U}|-1)\right)}\right) - 1.
\end{aligned}$$

As $s \in \mathbb{N}$ we use $s = \lfloor \log_2\left(\frac{\varepsilon n}{\log_e\left(\frac{\alpha}{1-\alpha}(|\mathcal{U}|-1)\right)}\right) - 1 \rfloor$ which concludes the proof. □

Algorithm 1 Algorithm PAD pads the input of party $P \in \{A, B\}$ such that the element with rank k is at the median position (part of FIND-RANKED-ELEMENT from [AMP10]).

Input: Data D_P, rank k, padding $\widehat{p}$
Output: Input padded to place k^{th}-ranked element at median position of the union of D_A, D_B
1: Sort D_P and retain only the k smallest values
2: Pad D_P with $+\infty$ until $|D_P| = k$
3: Pad D_P with $\widehat{p}$ until $|D_P| = 2^{\lceil \log_2(k) \rceil}$
4: **return** D_P

This is a *worst-case analysis* and a tighter upper bound can be obtained by using $|\mathcal{P}|$ instead of $|\mathcal{U}| - 1$. However, the size of $\mathcal{P}$ leaks information about D, hence, we refrain from using the tighter bound. Furthermore, we guarantee an accuracy of *at least* α, the actual accuracy can be even higher.

Lemma 1. With $s \in \mathcal{O}(\log(n) - \log\log(|\mathcal{U}|))$ the pruned data set's size is sublinear in the size of the data universe, i.e., $|D^s| = \frac{n}{2^s} \in \mathcal{O}(\log(|\mathcal{U}|))$[7].

2.5 Secure Sublinear Time DP Median Computation

First, we detail required subprotocols in Section 2.5.1. Then, we describe our full protocol in Section 2.5.2. In Section 2.5.3 we detail optimizations and present a runtime complexity analysis in Section 2.5.4. In Section 2.5.5 we prove the security of our protocol.

The notation "$\underline{A}$:" before an operation indicates that only party A performs the following operation, likewise for party B, and $L[i]$ denotes the element at index i in array L.

2.5.1 Subprotocols

Our protocol requires subprotocols, which we detail next.

Padding

Our protocol uses pruning developed by Aggarwal et al. [AMP10], which requires padding as a pre-processing step. Padding is described in Algorithm 1 where A calls PAD$(D_A, k, +\infty)$ and B calls PAD$(D_A, k, -\infty)$ with $k = \lceil (|D_A| + |D_B|)/2 \rceil$. Note that the data size of each party, i.e., $|D_A|, |D_B|$, can be hidden via additional padding.

Merge Implementation

The selection probabilities are computed on securely sorted, pruned data realized via oblivious merging from Huang et al. [HEK12].

For the merging implementation, as seen in Algorithm 2, we use the bitonic mergers as described in [HEK12, Section 5.1] which require a bitonic list as input, i.e., a list that is monotonically increasing then decreasing (or vice versa). Bitonic merging recursively splits the list in halves and compares and swaps elements such that every element of one half is greater than every element of the other half.

[7] We assume $n > \log(|\mathcal{U}|)$, as otherwise we do not require pruning and our input is already sublinear in the size of the universe.

Algorithm 2 Algorithm MERGE returns sorted $D^s = D^s_A \cup D^s_B$.

Input: Left index l, right index r, bitonic list D^s.
Output: Sorted D^s.

1: **return if** $r < l$
2: $m \leftarrow l + \frac{r-l}{2}$
3: **for** $i \leftarrow l$ **to** m **do**
4: $\quad e \leftarrow i + \left\lfloor \frac{r-l}{2} + 1 \right\rfloor$
5: $\quad$ Swap d^s_i with d^s_e **if** $d^s_i > d^s_e$
6: **end for**
7: MERGE$(l, m-1, D^s)$
8: MERGE$(m+1, r, D^s)$

Algorithm 3 Algorithm RANDOMDRAW with parties A, B based on [MM].

Input: Max. value M, lists of k nonces N_A, N_B from A, B.
Output: Uniform random integer in $[0, M)$

```
//Find most significant 1-bit in M, set following bits to 1 in mask
```
1: $c \leftarrow 0$
2: $mask \leftarrow 0$
3: **for** $i \leftarrow$ bitlength b **to** 1 **do**
4: $\quad c \leftarrow c$ OR i^{th} bit of M
5: $\quad i^{\text{th}}$ bit of $mask \leftarrow c$
6: **end for**

```
//Rejection sampling with abort
```
7: $s \leftarrow \perp$
8: **for** $i \leftarrow 1$ **to** k **do**
9: $\quad r \leftarrow N_A[i]$ XOR $N_B[i]$
10: $\quad r \leftarrow r$ AND $mask$
11: $\quad$ **if** $r < M$ **then**
12: $\quad\quad s \leftarrow r$
13: $\quad$ **end if**
14: **end for**
15: **if** $s = \perp$ **then**
16: $\quad$ **abort**
17: **end if**
18: **return** r

Random Draw

We implemented RANDOMDRAW, see Algorithm 3, with rejection sampling using efficient operations, namely XOR, OR, AND, comparison. Rejection sampling is unbiased, however, for a fixed input size of k nonces it might abort with probability at most 2^{-k}[8]. Rejection sampling (without

[8] We now consider the worst-case rejection rate, i.e., comparison $r < M$ in line 11 of Algorithm 3. Recall that r is the XOR of uniform random values, thus, each bit in r is uniform random as well. Masking ensures that at most the *mask* first bits of r are set, in effect reducing the size of r to *mask*. The number of set bits (i.e., bits with value 1) in M influences the rejection probability. The rejection rate is maximized if only one bit in M is set. Then, r is rejected with

abort) is used in Apple's macOS [MM]. For our evaluation in Section 2.6 we used $k = 20$.

An alternative to rejection sampling is a slightly biased sampling algorithm without abort requiring only one nonce instead of k per party: If the masked XOR of nonces (r) is larger than M one uses $r - M$ as the sampled output. We compared biased sampling with rejection sampling ($k = 20$) using the median of 20 runs for our largest circuit ($\varepsilon = 0.25, |D| = 2 \cdot 10^6$) with approximately 100 ms delay and 100 MBits/s bandwidth. Biased sampling required around 28k fewer gates and sent 400 KB less than rejection sampling with $k = 20$, which corresponds to a reduction in circuit size and communication of less than 1% for GC and around 3–4% for GC + SS. The runtime with biased sampling decreased by 2.2 seconds for GC (18.5% faster) but only by 0.18 seconds for GC + SS (2.6%). (For $k = 30$ an additional 44k gates and 600 KB are required compared to biased sampling, leading to similar runtimes as for $k = 20$.) Thus, we use rejection sampling as it is unbiased with only small impact on the runtime of GC + SS.

2.5.2 Protocol Description

Our protocol has four steps, denoted with (I)–(IV).

(I): Input Pruning (Algorithm 4)

Both parties prune their data sets D_A, D_B to D_A^s, D_B^s via [AMP10] using secure comparison realized with garbled circuits.

(II): Oblivious Merge & Secret Sharing (Algorithm 5)

The parties merge their pruned data D_A^s, D_B^s into sorted D^s via bitonic mergers from [HEK12] implemented with garbled circuits. Note that $D^s = \{d_0^s, \ldots, d_{|D^s|-1}^s\}$ is secret shared, i.e., A holds shares $\langle d_i^s \rangle_A$, B holds $\langle d_i^s \rangle_B$ for all $i \in I_{D^s}$.

(III): Selection Probability (Algorithm 6)

The parties compute $utility$, $weight$, and gap to produce shares of $mass$. Each party $P \in \{A, B\}$ now holds shares $\langle d_i^s \rangle_P$, $\langle gap(i) \rangle_P$ and $\langle mass(i) \rangle_P$ for all $i \in I_{D^s}$,

(IV): Median Selection (Algorithm 7)

The parties reconstruct all shares and select the differentially private median via inverse transform sampling realized with garbled circuits. First, they sample $d_j^s \in D^s$ based on $mass$. Then, they select the differentially private median $\widehat{m}$ at uniform random among the $gap(j)$ consecutive elements with the same utility as d_j^s.

2.5.3 Optimizations

To optimize the performance of the secure computation we utilize garbled circuits as well as secret sharing to use their respective advantages. E.g., multiplication of two b-bit values expressed as a Boolean circuit leads to a large circuit of size $\mathcal{O}(b^2)$ and is more efficiently done via secret

probability $1/2$ as all r with 0 at position $mask$ are accepted ($r < M$), while the other half is rejected. Increasing the number of set bits in M decreases the rejection rate (as more r can be smaller than M). Thus, the rejection probability per sample r is at most $1/2$.

Algorithm 4 PRUNE prunes D_A, D_B to D_A^s, D_B^s via [AMP10].

Input: Data D_A from A, D_B from B, pruning steps s, median rank $k = \lceil (|D_A| + |D_B|)/2 \rceil$.

Output: A has pruned data D_A^s, likewise B has D_B^s.

1: $\underline{A}$: $D_A^0 \leftarrow \text{PAD}(D_A, k, +\infty)$ `//Algorithm 1`
2: $\underline{B}$: $D_B^0 \leftarrow \text{PAD}(D_B, k, -\infty)$
3: **for** $i \leftarrow 0$ **to** $s-1$ **do**
4: $\quad \underline{A}$: $m_A \leftarrow$ median of D_A^i
5: $\quad \underline{B}$: $m_B \leftarrow$ median of D_B^i
6: $\quad c \leftarrow m_A < m_B$
7: $\quad \underline{A}$: $D_A^{i+1} \leftarrow$ upper half of D_A^i **if** $c = 1$ **else** lower half
8: $\quad \underline{B}$: $D_B^{i+1} \leftarrow$ lower half of D_B^i **if** $c = 1$ **else** upper half
9: **end for**

Algorithm 5 MERGEANDSHARE merges D_A^s, D_B^s into sorted D^s via [HEK12] and secret shares it.

Input: Pruned data D_A^s from A in ascending order, array $\langle D^s \rangle_A$ of $2|D_A^s|$ random values in $\mathbb{Z}_{2^{64}}$ from A, D_B^s from B sorted in descending order.

Output: A has secret shares $\langle D^s \rangle_A$ of sorted union of pruned data, resp. B has $\langle D^s \rangle_B$.

1: $D^s \leftarrow D_A^s$ appended with D_B^s
2: $\text{MERGE}(0, |D^s| - 1, D^s)$ `//Algorithm 2`
3: $\langle D^s \rangle_B \leftarrow D^s - \langle D^s \rangle_A \mod 2^{64}$
4: **return** $\langle D^s \rangle_B$ **to** B

Algorithm 6 SELECTIONPROBABILITY computes the probabilities for the median utility.

Input: Secret shares $\langle D^s \rangle_A$ from A, resp. $\langle D^s \rangle_B$ from B, of the sorted data D^s, and number k of nonces.

Output: A holds secret shares $\langle gap \rangle_A$ of gaps and $\langle mass \rangle_A$ of probability masses, also nonces N_A^1, N_A^2; likewise party B has $\langle gap \rangle_B, \langle mass \rangle_B, N_B^1, N_B^2$.

1: $\underline{A}$: $\langle D^s \rangle_A \leftarrow (0, \langle D^s \rangle_A, 0)$
2: $\underline{B}$: $\langle D^s \rangle_B \leftarrow (\min(\mathcal{U}), \langle D^s \rangle_B, \max(\mathcal{U}))$
3: **each** party $P \in \{A, B\}$ **does**
4: $\quad$ Define arrays $\langle mass \rangle_P, \langle gap \rangle_P$ of size $|D^s|$
5: $\quad$ **for** $i \leftarrow 0$ **to** $|D^s| - 1$ **do**
6: $\quad\quad utility \leftarrow \begin{cases} i - \frac{|D^s|}{2} + 1 & \textbf{if } i < \frac{|D^s|}{2} \\ \frac{|D^s|}{2} - i & \textbf{else} \end{cases}$
7: $\quad\quad weight \leftarrow \exp(\varepsilon \cdot utility)$
8: $\quad\quad \langle gap[i] \rangle_P \leftarrow \begin{cases} \langle d_{i+1}^s \rangle_P - \langle d_i^s \rangle_P & \textbf{if } i < \frac{|D^s|}{2} - 1 \\ \langle 1 \rangle_P & \textbf{if } i = \frac{|D^s|}{2} - 1 \\ \langle d_i^s \rangle_P - \langle d_{i-1}^s \rangle_P & \textbf{else} \end{cases}$
9: $\quad\quad t \leftarrow \langle mass[i-1] \rangle_P$ **if** $i > 0$ **else** 0
10: $\quad\quad \langle mass[i] \rangle_P \leftarrow t + weight \cdot \langle gap[i] \rangle_P$
11: $\quad$ **end for**
12: $\quad$ Draw lists of k nonces N_P^1, N_P^2 from $[0, \max(\mathcal{U}) - \min(\mathcal{U})]$
13: **end each**

Algorithm 7 MEDIANSELECTION selects the median via inverse transform sampling.

Input: Secret shares $\langle gap \rangle_A$ of gaps, $\langle mass \rangle_A$ of probability masses, and $\langle D^s \rangle_A$ of A's (pruned) data, also lists of nonces N_A^1, N_A^2 from A; resp. $\langle gap \rangle_B, \langle mass \rangle_B, \langle D^s \rangle_B, N_B^1, N_B^2$ from B.

Output: Differentially private median $\widehat{m}$ of $D_A \cup D_B$.

1: $R \leftarrow \langle mass\,[|D^s|-1] \rangle_A + \langle mass\,[|D^s|-1] \rangle_B \mod 2^{64}$
2: $r \leftarrow \text{RANDOMDRAW}(R+1, N_A^1, N_B^1)$ `//Algorithm 3`
3: Initialize $j \leftarrow -1$ and define d, g
4: **for** $i \leftarrow 0$ **to** $|D^s|-1$ **do**
5: $\quad e \leftarrow \langle d_i^s \rangle_A + \langle d_i^s \rangle_B \mod 2^{64}$ `//Recombine shares`
6: $\quad gap \leftarrow \langle gap[i] \rangle_A + \langle gap[i] \rangle_B \mod 2^{64}$
7: $\quad mass \leftarrow \langle mass[i] \rangle_A + \langle mass[i] \rangle_B \mod 2^{64}$
8: $\quad$ **if** $r < mass$ **and** $j = -1$ **then**
9: $\quad\quad d \leftarrow e$; $g \leftarrow gap$; $j \leftarrow i$
10: $\quad$ **end if**
11: **end for**
12: $x \leftarrow \text{RANDOMDRAW}(g, N_A^2, N_B^2)$
13: $\widehat{m} \leftarrow \begin{cases} d+x & \textbf{if } j < \frac{|D^s|}{2} - 1 \\ d-x & \textbf{else} \end{cases}$
14: **return** $\widehat{m}$ **to** A, B

sharing. On the other hand, comparison is more efficient with garbled circuits. Algorithms 5, 6 are implemented with garbled circuits. In Algorithm 4 only line 6 requires garbled circuits, the rest is either data-independent or executed locally. Secret shares, denoted with $\langle \cdot \rangle$, are created in Algorithm 5, used in Algorithm 6, and recombined in Algorithm 7. Furthermore, we compute the required exponentiations in Algorithm 6 line 7 without any secure computation. Next we reiterate portions of Section 2.4.2 but in the new context of secure computation.

Sorting via Garbled Circuits

Our utility definition requires the data to be sorted which inherently relies on comparisons. Comparisons are more efficiently implemented in binary circuits than arithmetic circuits, hence, we use the former. We leverage that D_A^s and D_B^s are already sorted and merge them instead of sorting the union. Oblivious merging of two lists of n sorted b-bit elements only requires $2bn\log(n)$ binary gates whereas oblivious sorting requires $\Theta(n\log(n))$ with a large constant factor [HEK12]. We use *bitonic mergers* from Huang et al. [HEK12] which require a bitonic list as input, i.e., a list that monotonically increases then decreases (or vice versa). We can generate a bitonic list by appending D_A^s sorted in ascending order with D_B^s sorted in descending order (Algorithm 5 line 1).

Exponentiation without Secure Computation

To compute the probabilities for $i \in I_{D^s}$ we require exponentiations of the form $\exp(\varepsilon \cdot utility(i))$. Note that none of the arguments are secret, since ε is a public parameter and we defined $utility$ to not require data access. Therefore, we are able to compute the required exponentiations without any secure computation.

Addition and Multiplication via Secret Sharing

We want to compute the probability mass $weight(i) \cdot gap(i)$ which requires two operations: subtractions over secret data D^s to compute gap and multiplication of public values ($weight$), with secret values (gap). Both operations are more efficiently implemented with secret sharing.

Selection via Garbled Circuits

The median selection is realized with inverse transform sampling which is better suited for garbled circuits as it requires comparisons. We draw a random r via nonces (see Appendix 2.5.1) and compute the first index $j \in I_{D^s}$ such that the probability mass is larger than r: $mass(j) > r$ (line 8 in Algorithm 7). Note that we do not sample r from $[0,1]$ but from $[0,R]$ where $R = mass(|D^s|-1)$, i.e., the normalization factor from Equation (2.1). This allows us to use the unnormalized probabilities and eliminates divisions used in normalization. In the final step, we select the differentially private median at uniform random among the $gap(j)$ consecutive elements with the same utility (and thus probability) as d^s_j (line 13 in Algorithm 7).

2.5.4 Runtime Complexity Analysis

Step (I), requires $s \in \mathcal{O}(\log n - \log\log|\mathcal{U}|)$ comparisons (see Theorem 4). Step (II) requires $2b|D^s|\log|D^s|$ binary gates [HEK12] for $|D^s|$ elements with bit length b. Steps (III) and (IV) require $\mathcal{O}(|D^s|)$ operations each. Since $|D^s| \in \mathcal{O}(\log|\mathcal{U}|)$ (Lemma 1), our overall runtime is

$$\mathcal{O}(\max\{\log n - \log\log|\mathcal{U}|, \log|\mathcal{U}| \cdot \log\log|\mathcal{U}|\}),$$

which is sublinear in n for $n > \log|\mathcal{U}|^{\log|\mathcal{U}|+1}$, and sublinear in $|\mathcal{U}|$ otherwise.

2.5.5 Security

We combine different secure computation techniques in the *semi-honest model* introduced by [Gol09] where corrupted protocol participants do not deviate from the protocol but gather everything created during the run of the protocol. Our protocol consists of multiple subroutines realized with secure computation. To analyze the security of the entire protocol we rely on the well-known *composition theorem* [Gol09, Section 7.3.1]. Basically, a secure protocol that uses an ideal functionality (a subroutine provided by a trusted third party) remains secure if the ideal functionality is replaced with a secure computation implementing the same functionality. We consider PRUNE-neighboring data sets (Definition 3), i.e., neighboring data sets with the same pruning result.

Theorem 5 (Security)**.** Our protocol securely implements the ideal functionality of differentially private median selection via the steps PRUNE, MERGEANDSHARE, SELECTIONPROBABILITY and MEDIANSELECTION in the semi-honest model.

Proof. First, we show that the partial execution of PRUNE is secure based on a simulation argument [AMP10]. Then, we use the composition theorem to analyze the security of our protocol: We define required ideal functionalities, show how they map to our garbled circuit implementation (steps (I), (II), (IV)), and how it combines with secret sharing (step (III)).

Pruning. Aggarwal et al. [AMP10] developed the input pruning we utilize and give a simulation-based security proof only using comparisons as ideal functionality. PRUNE, a partial execution of [AMP10], allows the same simulation argument.

Algorithm 8 Algorithm SIMULATEPRUNING simulates the secure k^{th}-ranked element computation from [AMP10].

Input: Parameter element rank k, real execution result m and iteration count j. Note that D_A is known to A and all items in $D_A \cup D_B$ are distinct.

Output: Simulation of running the protocol for finding the k^{th}-ranked element m in $D_A \cup D_B$.

1: A initializes $D_A^1 \leftarrow \text{PAD}(D_A, k, +\infty)$ `//Appendix 2.5.1`
2: **for** $i \leftarrow 0$ **to** $j-1$ **do**
3: A computes $m_A \leftarrow$ median of D_A^i
4: Secure comparison result c is set to 1 if $m_A < m$ (i.e., $m_A < m_B$) otherwise it is 0
5: A sets $D_A^{i+1} \leftarrow$ upper half of D_A^i if $c = 1$ otherwise it is the lower half
6: **end for**
7: The final secure comparison result c is set to 1 if $m_A < m$ and else it is 0

Aggarwal et al. [AMP10] prove the security of their exact k^{th}-ranked element computation in the semi-honest model by showing that A (similarly B) can simulate the secure protocol given its own input D_A, and the value m of the k^{th}-ranked element. We reproduce their simulation in the following as we use the same argument with small modifications.

The simulation executed by A (similarly B) in [AMP10] is detailed in Algorithm 8. If the data D_A contains duplicates, $\lceil \log_2 |D_A| \rceil + 1$ bits are added to the binary representation of each element to make it unique as required for the simulation. E.g., A adds for each element the bit 0 followed by the rank of the element in the least significant bit positions. B follows the same procedure using 1 instead of 0. These bits are removed from the final output.

Aggarwal et al. [AMP10] execute the simulation as SIMULATEPRUNING$(k, m, \lceil \log_2(k) \rceil)$, i.e., full pruning until only one element remains. Lemma 2 from [AMP10] states that the transcript of the real execution and the simulated execution are equivalent. Additionally, the state information, i.e., pruned data D_A^i, that A has at each iteration i is the same as well.

In our protocol we do not perform the full execution, i.e., only s iterations. We do not know the exact value m, however, A knows its state D_A^s at the final step and we use median of D_A^s instead of m. Altogether, we call the simulation with SIMULATEPRUNING$(k, \text{median of } D_A^s, s)$.

We now show by contradiction that our simulation outputs the correct comparison results. Assume $c = 1$, i.e., $m_A < m_B$, at iteration i in our real execution but our simulation outputs 0, i.e., $m_A \geq$ median of D_A^s. Then D_A^{i+1} is the lower half of D_A^i and only elements smaller than or equal to $m_A =$ median of D_A^i remain in D_A^{i+1} and thus in D_A^s. In other words, for $x \in D_A^{i+1}$ we have $x \leq m_A$ and due to $D_A^s \subseteq D_A^{i+1}$ we have median of $D_A^s < m_A$. However, this contradicts $m_A \geq$ median of D_A^s, i.e., output 0. Analogously, we find a contradiction if $c = 0$ in our real execution but 1 in the simulation.

Ideal Functionalities. For the interactive computation we require the following ideal functionalities:

- $c \leftarrow \text{SECURECOMPARE}^{\text{ideal}}(m_A; m_B)$.

  In step (I) the ideal functionality on input m_A, m_B, i.e., median from A, B respectively, outputs the result of comparison $m_A < m_B$ as bit c to both parties.

- $\langle D^s \rangle_A, \langle D^s \rangle_B \leftarrow \text{MERGEANDSHARE}^{\text{ideal}}(D_A^s; D_B^s)$.

  In step (II) the ideal functionality receives as input the pruned data D_A^s, D_B^s from A, B respectively, and outputs the sorted, merged data as secret shares, i.e., $\langle D^s \rangle_A, \langle D^s \rangle_B$ is output

to A, B respectively.

- $\widehat{m} \leftarrow \text{MEDIANSELECTION}^{\text{ideal}}(\langle gap \rangle_A, \langle mass \rangle_A, \langle D^s \rangle_A; \langle gap \rangle_B, \langle mass \rangle_B, \langle D^s \rangle_B)$.

  In step (IV) party A inputs $\langle gap \rangle_A, \langle mass \rangle_A, \langle D^s \rangle_A$, party B inputs $\langle gap \rangle_B, \langle mass \rangle_B, \langle D^s \rangle_B$ and the ideal functionality outputs the DP median $\widehat{m}$ to both.

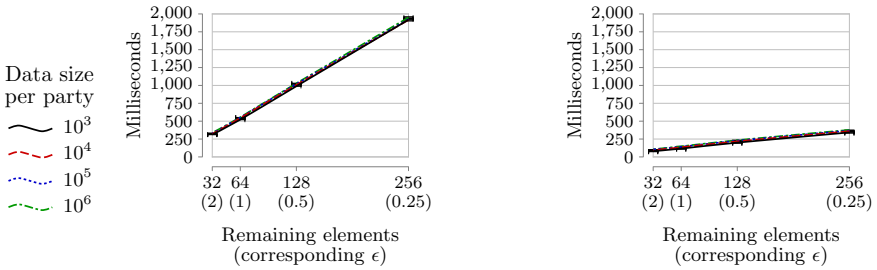
Step (III), SELECTIONPROBABILITY, performs local computations without interaction, and does not require any ideal functionality. We realize $\text{SECURECOMPARE}^{\text{ideal}}$ with garbled circuits in Algorithm 4 line 6. The ideal functionality $\text{MERGEANDSHARE}^{\text{ideal}}$, from merging step (II), is implemented as MERGEANDSHARE in Algorithm 5 with garbled circuits. Note that A provides the randomness for the secret sharing, i.e., $\langle D^s \rangle_A$ as additional input which is not required by the ideal functionality. Garbled circuits are also used in the selection step (IV), where $\text{MEDIANSELECTION}^{\text{ideal}}$ is implemented as MEDIANSELECTION in Algorithm 7. Additionally, to the input mentioned for the ideal functionality, the parties also provide nonces as a source of randomness. We rely on the established security proofs for garbled circuits in the semi-honest model provided by Lindell and Pinkas [LP09]. Outputs of (II), (III) are intermediate states of our interactive computation. As noted in [Gol09, Section 7.1.2.3] such state can be maintained securely among the computation parties in a secret sharing manner. For security proofs of secret sharing we refer to [PBS12] and for security proofs for converting between garbled circuits and secret sharing we refer to [DSZ15]. Altogether, the execution of $\text{PRUNE}^{\text{ideal}}$, $\text{MERGEANDSHARE}^{\text{ideal}}$, SELECTIONPROBABILITY, and $\text{MEDIANSELECTION}^{\text{ideal}}$ constitute the ideal functionality for differentially private median. Utilizing the composition theorem and [Gol09, Section 7.1.2.3] we replace the ideal functionality with secure implementations PRUNE, MERGEANDSHARE, MEDIANSELECTION and secret share the intermediate states. □

2.6 Evaluation

Our implementation is written in C/C++ using the mixed-protocol framework *ABY* developed by Demmler et al. [DSZ15]. We chose ABY as it supports secure two-party computation based on arithmetic sharing and Yao's garbled circuits and provides efficient conversion between them. We implemented two versions of our protocol

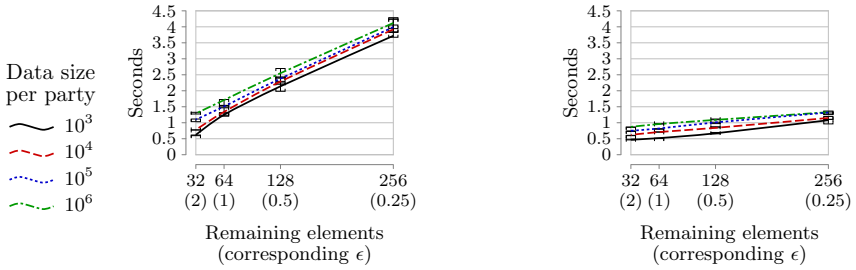
- GC, with garbled circuits,
- GC + SS, with garbled circuits as well as secret sharing,

to show that using a mixed-protocol, which requires additional conversion between the schemes, is still more efficient than only utilizing garbled circuits. For evaluation we used the Open Payments 2017 data set from the Centers for Medicare & Medicaid Services (CMS) [Cen17]. The CMS collects all payments made to physicians from drug or medical device manufacturers as required by the Physician Payments Sunshine Act. We evaluated different numbers of remaining elements after pruning (i.e., different sizes of D^s) which is inversely proportional to the privacy parameter ε as the number of pruning steps depends on it (see Theorem 4). We used an accuracy value of 0.9999 to determine the number of pruning steps. We ran the evaluation on AWS t2.medium instances with 2GB RAM and 4 vCPUs (where vCPU count roughly translates to thread count). As garbled circuits and pruning are interactive protocols they are influenced by network delay and bandwidth, therefore, we evaluated our protocol in real networks between different AWS



(a) GC (b) GC + SS

Figure 2.5: Runtime without network delay and 1 GBits/s bandwidth (LAN).



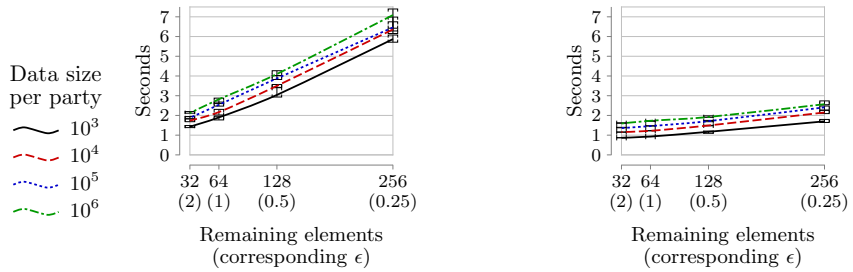
(a) GC (b) GC + SS

Figure 2.6: Runtime for ∼12 ms RTT, ∼430 MBits/s (Ohio and N. Virginia).

regions with round trip times (RTT) of none (LAN), 12 ms (Ohio–N. Virginia), 25 ms (Ohio–Canada), and 100 ms (Ohio–Frankfurt), with bandwidths of 1 GBits/s, 430 MBits/s, 160 MBits/s and 100 MBits/s respectively.

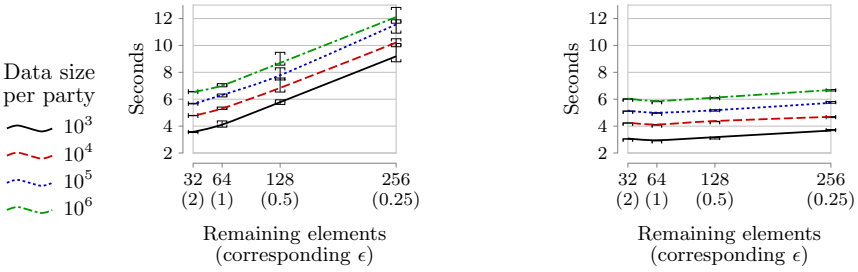
2.6.1 Runtime

We evaluated the runtime of GC and GC + SS, which includes setup time (OT extensions, garbling) and online time in seconds (or milliseconds in the LAN setting). The runtime evaluations with increasing delays and decreasing bandwidths are presented in Figure 2.5–2.8. In each figure we plotted different data set sizes $|D_A| = |D_B| = |D|/2 \in \{10^3, 10^4, 10^5, 10^6\}$ to show that our protocol scales with increasingly larger data sets. The runtime is the median of 20 runs and the 25$^{\text{th}}$- as well as 75$^{\text{th}}$-percentile are indicated with brackets. The runtime plots for GC and GC + SS have the same scale (and are grouped side-by-side) to allow for an easier comparison between the two. The advantage of GC + SS over GC is most obvious in the LAN setting, where the runtime for GC + SS, see Figure 2.5b, is always below that of GC, see Figure 2.5a. The same is true for modest network delay as can be seen by comparing Figure 2.6b with Figure 2.6a. For network delay of up to 100 ms with 100 MBits/s bandwidth GC + SS is still faster than GC but less so for 32 remaining elements ($\varepsilon = 2$), as shown in Figure 2.7 and 2.8. The reason for GC + SS being not much faster is the increased number of interactive pruning steps required to reach this number of remaining elements. Also, the number of additional garbled circuits to go from GC + SS to GC is smaller for few remaining elements (see Figure 2.11a), so that the pruning has more impact. Even for millions of records GC + SS has a runtime of less than 2.6 seconds with 25 ms network delay (Figure 2.7b) and less than 7 seconds for 100 ms delay (Figure 2.8b).



(a) GC (b) GC + SS

Figure 2.7: Runtime for ∼25 ms RTT, ∼160 MBits/s (Ohio and Canada).



(a) GC (b) GC + SS

Figure 2.8: Runtime for ∼100 ms RTT, ∼100 MBits/s (Ohio and Frankfurt).

Index		$j-1$	j	$j+1$	
D_B	$\cdots$	d_{j-1}	d_j	d_{j+1}	$\cdots$
$D_B \setminus \{x\}$	$\cdots$	d_j	d_{j+1}	d_{j+2}	$\cdots$
$D_B \cup \{x\}$	$\cdots$	$[d_{j-3}, d_{j-2}]$	$[d_{j-2}, d_{j-1}]$	d_{j-1}	$\cdots$

Figure 2.9: Neighbors of D_B in relation to comparison index j used by PRUNE (values highlighted in gray). Neighbors are D_B with a value $x \in D_B$ removed or $x \in \mathcal{U}$ added, illustrated for $x < d_j$. All data sets are sorted.

2.6.2 PRUNE-neighboring

Recall, PRUNE compares the *sorted, padded* data D_A, D_B at some fixed index j in each pruning step, and a neighbor is D_B with an element x removed or added. As Figure 2.9 illustrates, comparing a neighbor at index j is similar to using the original D at an adjacent index. Thus, neighbors are likely PRUNE-neighbors when the data contains multiple duplicates or is dense (no large gaps between values) and less so for sparse, unique data. In more detail, we first consider $x < d_j$ where d_j denotes the value of D_B at index j. Let the data be padded to some fixed size. Then, removing x from D_B "shifts" values larger than x to the left whereas adding x can shift smaller values to the right in the sorted data. Removing $x \in D_B$ leads to a single shift left, i.e., PRUNE uses d_{j+1} instead of d_j. For addition at most two right shifts can occur as we now have to consider $x \in \mathcal{U}$ instead of $x \in D_B$. Adding $x \in [d_{j-2}, d_{j-1}]$ places it at index j in the sorted neighbor. Thus, in the worst-case for addition, PRUNE uses d_{j-2} instead of d_j. Note that adding/removing $x \geq d_j$ affects only positions larger than j, and all such neighbors are PRUNE-neighbors for this index.

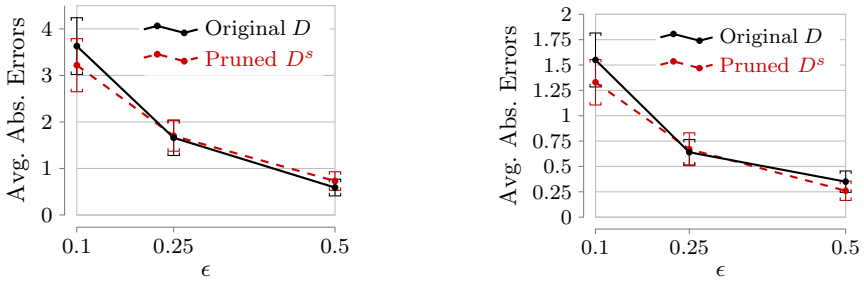
Table 2.3: **Minimum changes** (worst-case) in D_B to sample a neighbor that is not a PRUNE-neighbor w.r.t. D_A. Evaluated for 52 000 neighbors (all combinations of up to 50 removals and 50 additions with 20 samples per combination). Each row shows the minimum changes for $\varepsilon = 1 \mid \varepsilon = 2$ and $\hat{>}_{100}$ indicates none were found for up to 100 changes.

D_A \ D_B	*Wages [Soo18]*	*Transactions [ULB18]*	*Times [ULB18]*	*Payments [Cen17]*	*Weights [Kag18]*	*Quantities [Kag18]*
Wages [Soo18]	$\hat{>}_{100} \mid 18$	$\hat{>}_{100} \mid 14$	$12 \mid 12$	$22 \mid 22$	$\hat{>}_{100} \mid 12$	$46 \mid 21$
Transactions [ULB18]	$65 \mid 65$	$8 \mid 8$	$\hat{>}_{100} \mid 20$	$37 \mid 30$	$36 \mid 36$	$23 \mid 23$
Times [ULB18]	$\hat{>}_{100} \mid 22$	$33 \mid 18$	$6 \mid 6$	$\hat{>}_{100} \mid 13$	$\hat{>}_{100} \mid 21$	$25 \mid 25$
Payments [Cen17]	$28 \mid 28$	$\hat{>}_{100} \mid 11$	$\hat{>}_{100} \mid \hat{>}_{100}$	$6 \mid 6$	$\hat{>}_{100} \mid 41$	$\hat{>}_{100} \mid 13$
Weights [Kag18]	$\hat{>}_{100} \mid 43$	$34 \mid 33$	$4 \mid 4$	$33 \mid 33$	$\hat{>}_{100} \mid 21$	$48 \mid 19$
Quantities [Kag18]	$30 \mid 30$	$\hat{>}_{100} \mid 25$	$\hat{>}_{100} \mid 12$	$\hat{>}_{100} \mid 9$	$14 \mid 14$	$14 \mid 14$

Also, if the original comparison (of D_A, D_B at j) is true, then removing $x < d_j$ produces the same result in PRUNE (neighbor has an even larger value at j). Likewise if it is false and we add x. To empirically verify that PRUNE-neighboring (Definition 3) is not too restrictive we evaluated multiple columns from real-world data sets [Cen17, Kag18, Soo18, ULB18], and found that all neighbors are also PRUNE-neighbors. To illustrate our evaluation methodology one can imagine the neighboring definition in differential privacy (DP) as a graph. Each database is a vertex and if two data sets are neighbors they are connected by an edge. The common neighboring definition in DP (adding/removing one element) results in a graph. PRUNE-neighboring is a restriction on that graph in the sense that it removes certain edges, similar constraints on the input databases are considered in [BSRW17, HMFS17]. Any neighboring database considered in our IND-CDP-2PC security definition must be in a connected component of the neighboring graph where all nodes have the same output of the PRUNE-function. The result of the PRUNE steps in our protocol determines the connected component the other party's database is DP in. In that sense DP with PRUNE-neighboring cannot be violated by any adversary. Any choice of inputs by party A will lead to one (but different) connected component for the DP of B's database, i.e., B's database will always remain differentially private. We empirically showed that PRUNE-neighboring is not too restrictive, i.e., it does not remove too many edges and make the resulting connected component too small. We sampled edges from the neighboring graph resulting from the common definition on real-world data sets [Cen17, Kag18, Soo18, ULB18] using the following method: Given a real-world database for B, an element to be added or removed chosen by A (note that A must choose before knowing the result), and a step in the protocol does there exist any neighbor for B's database that is excluded by the PRUNE-neighboring definition. For up to 16 consecutive pruning steps (the maximum according to Theorem 4 for our highest evaluated parameters $\varepsilon = 2$, and accuracy of 0.9999), we found none. Given that the connectivity in the neighboring graph is high, this implies that the connected component is expected to remain large.

Group privacy extends the neighboring definition from including (or excluding) a single value to multiple values. Therefore, to quantify group privacy we consider *multiple* changes and provide a worst-case analysis for PRUNE-neighboring: Table 2.3 shows the *minimum* changes required to produce a neighbor that is not also a PRUNE-neighbor[9]. We evaluated 52 000 neighbors (all

[9] Some values are the same for $\varepsilon \in \{1,2\}$ as we only report the minimum number of changes over all pruning steps.



(a) Credit card data [ULB18], first 10^5 payment records in Cents.

(b) Walmart supply chain data [Kag18], 175k shipment weights as integers.

Figure 2.10: Absolute error averaged over 100 runs with and without pruning.

combinations of up to 50 removals and 50 additions with 20 samples per combination) for each of the 36 ways to distribute the data between two parties (6 data sets from [Cen17, Kag18, Soo18, ULB18] distributed between 2 parties). PRUNE-neighboring provides only limited group privacy for the largest number of pruning steps ($\varepsilon = 2$). However, for our strongest privacy guarantee $\varepsilon = 0.25$ we found changes leading to violations in only 2 from 36 data set combinations, requiring at least 12 changes. Sequential composition is still supported as the result of our protocol is the median selected by the exponential mechanism which can be used as input for another (DP) mechanism. (Parallel composition, running our protocol on multiple subsets of the data at once, outputs multiple median values of these subsets.)

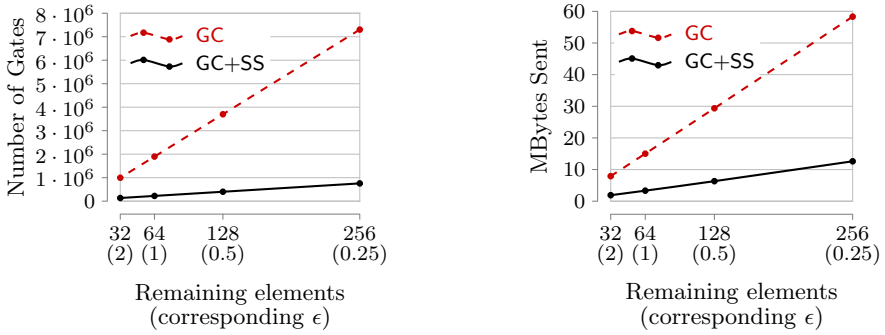
2.6.3 Precision & Absolute Error

Our implementation uses fixed-point numbers (see Section 2.3). As probabilities are floating point numbers we evaluated the loss of decimal precision of our secure implementation compared to a floating point operation with access to unprotected data [Cen17]. For the maximum evaluated number of remaining elements, i.e., 256 (corresponding to $\varepsilon = 0.25$), the difference for all elements combined was less than $6.5 \cdot 10^{-15}$.

Pruning preserves the elements closest to the median and the absolute error compared to the original data is small. We evaluated the absolute error, i.e., actual median versus DP median, for the exponential mechanism on original data and pruned data: Figure 2.10 shows the average over 100 runs, where brackets indicate the 95% confidence interval. Before pruning the data was randomly split between both parties. Our evaluation shows the absolute error decreases by 3% on average over all evaluated $\varepsilon \in \{0.1, 0.25, 0.5\}$. However, this is within the margin of error, since the confidence intervals for pruned data overlap with original data's confidence intervals.

2.6.4 Circuit size & Communication

We only report circuit size and communication for 10^6 records as smaller data sizes (i.e., fewer pruning steps) do not noticeably reduce the numbers (recall, a pruning step consists of a single comparison). The number of garbled gates for GC and GC + SS depends on the number of remaining elements and is visualized in Figure 2.11a. GC requires an order of magnitude more gates as GC + SS since GC requires larger circuits for arithmetic operations whereas GC + SS avoids the need for this additional circuit complexity. The communication cost, measured in megabytes per number of remaining elements, can be found in Figure 2.11b. We do not distinguish between



(a) Number of garbled circuit gates.

(b) Total megabytes sent.

Figure 2.11: Circuit size and communication for GC vs. GC + SS.

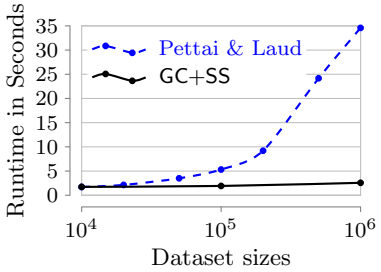


Figure 2.12: Runtime of GC + SS (∼25 ms RTT and ∼160 MBits/s, 256 remaining elements, $\varepsilon = 0.25$) vs. Pettai and Laud [PL15] (LAN).

(precomputed) setup and online phase and present the total number of megabytes sent. Whereas GC sends about 15 megabytes for 64 remaining elements ($\varepsilon = 1$), GC + SS requires less than that even for 256 remaining elements ($\varepsilon = 0.25$) as fewer gates have to be garbled and evaluated.

2.6.5 Comparison to Related Work

Pettai and Laud [PL15] compute differentially private analytics on distributed data via secret sharing for three parties, whereas we optimize our protocol for rank-based statistics of two parties and also use garbled circuits.[10] Both parties learn the PRUNE-neighborhood (for large data sets requiring pruning), but the median output can be shared (or output to only a single party) and processed further. Pettai and Laud evaluated their median computation with 48GB RAM and a 12-core 3GHz CPU in a LAN. We, on the other hand, used a comparatively modest setup (t2.medium instances with 2GB RAM, 4vCPUs) and evaluated in multiple WANs. A comparison of our protocol (with ∼25 ms delay, ∼160 MBits/s) and [PL15] (in a LAN) is visualized in Figure 2.12. Their median computation requires 34.5 seconds for 10^6 elements in a LAN. Our protocol runs in less than 2.6 seconds with twice as many elements even with network delay and restricted bandwidth.

[10] Note that 3-party computation on secret shares are usually faster than cryptographic 2-party computations [ABPP16].

2.7 Related Work

We describe related work combining secure computation with differential privacy, outline alternatives to reduce the size of the data universe, and discuss other work that computes the differentially private (DP) median.

Secure Computation and DP

Dwork et al. [DKM+06] first mentioned that differential privacy combines well with secure computation. E.g., secure computation of DP sums is easily achieved via additive noise (see [GX17] for an overview). It was shown in [GKM+16] that some distributed DP protocols (e.g., XOR computation) can only achieve optimal accuracy when combined with secure computation. We utilize the iterative pruning from Aggarwal et al. [AMP10] as it is a basis for more efficient secure computation protocols as shown in [SV15]. (Not all protocols can utilize this approach, e.g., it is not applicable when only one party learns the output [BIP18]). Naor et al. [NPR19] use secure two-party computation to find differentially private heavy hitters (e.g., to blacklist frequently used passwords) in the local model. They also consider malicious adversaries that try to skew the frequency. We, on the other hand, simulate the more accurate central model in the local model to find the DP median in the semi-honest model. For functions that are not robust to potentially large noise, e.g., the median, a specific value from a data universe, the exponential mechanism, developed by McSherry and Talwar [MT07], is the better choice [LLSY16]. The exponential mechanism defines a probability distribution over all possible output values. Eigner et al. [EKM+14] implement the exponential mechanism in secure multiparty computation for semi-honest and malicious parties. However, they are linear in the size of the data universe: 3 semi-honest parties require 42.3 seconds to sample a universe of size 5 in a LAN on a machine with 32GB RAM and 3.2GHz CPU. Our protocol is sublinear in the size of the data universe, requiring less than 500 milliseconds for millions of elements in a LAN with less powerful hardware (see Figure 2.5b). Efficiently sampling the distribution defined by the exponential mechanism is non-trivial [DR14], thus, a reduction of the sampling space is considered by [BDB16, GLM+10, LLSY16, PL15].

Pruning and Reduction

Gupta et al. [GLM+10] suggest pruning the set of outputs for combinatorial problems from exponential to polynomial size and sample it with the exponential mechanism. We follow a different approach based on [AMP10]. Another technique divides $\mathcal{U}$ into equal-sized ranges, selects a range with the exponential mechanism and samples a range element at uniform random [LLSY16]. However, any element in the selected range is equally likely to be output independent of its utility. Our protocol samples the median only among elements with the same utility which is exponentially more likely to select elements closer to the actual median. Pettai and Laad [PL15] define algorithms for privacy-preserving analytics. They securely compute the DP median with three parties but chose not to optimize their computation for the exponential mechanism and instead use the sample-and-aggregate mechanism [NRS07]. The sample-and-aggregate mechanism divides the output in multiple equal-sized ranges, selects from each range the element closest to the median and returns a noisy average of these elements. However, the exponential mechanism, which we securely implement for the median utility function, selects an actual universe element and not a noisy approximation. The authors of [PL15] also apply input pruning and replace half of the excluded values with a small (resp. large) constant. They mention that this does not always preserve

the median. Blocki et al. [BDB16] use a relaxed exponential mechanism to sample a DP password frequency list in the central model. They allow a negligible error δ, i.e., they only sample the exponential mechanism correctly with probability $1-\delta$, which improves sampling from (potentially) exponential time to $\mathcal{O}(|D|^{1.5}/\varepsilon)$. However, they require full access to the data D in clear.

Differentially Private Median

As mentioned above, Pettai and Laud [PL15] also securely compute the DP median. Their work is more general, supporting multiple DP statistics over secret-shared data, whereas we optimized our protocol for rank-based DP statistics (e.g., p^{th}-percentile, median) in a two-party setting without powerful hardware. Their protocol requires 34.5 seconds for a data size of 10^6 in a LAN [PL15, Figure 1] whereas our protocol runs in less than 500 ms with twice as many elements in a LAN (Figure 2.5b) and is still 13 times faster in a WAN as [PL15] in a LAN (Figure 2.12). Median computation has also been considered in the DP query framework PINQ, developed by McSherry and Talwar [McS09], which requires a trusted third party. Smooth sensitivity, presented in [NRS07], analyzes the data to provide instance-specific additive noise. Yet, when smooth sensitivity is high, it still provides less accuracy than the exponential mechanism (see Section 2.2). Also, computing the exact sensitivity itself is not trivial and requires access to the entire, sensitive data set. Another approach from Dwork and Li [DL09] considers the statistical setting, where data are actually i.i.d. samples from a distribution. Their approach requires additive noise proportional to the scale of the data (approximated via interquartile range), i.e., potentially large noise, whereas our result is independent of the scale. Smith et al. [STU17] compute the DP median in the local model and achieve optimal error bounds without relying on secure computation and even avoid interaction; however, the local model's accuracy is limited compared to the central model ($\Omega(\sqrt{n})$ vs. $\mathcal{O}(1)$ for n parties [HKR12]). They approximate for each party the count of elements in all subintervals of a range, structured as nodes in a tree. A server combines these noisy counts to learn the DP median. Hsu et al. [HKR12] consider approximate counts for heavy hitters and say an algorithm is α-accurate if the returned universe element occurs with frequency that differs at most by an additive α from the true heavy hitter. They show that the lower bound for accuracy in the local model (the setting of [STU17]) is $\Omega(\sqrt{n})$ for n parties, whereas the central model, which we simulate via secure computation of the exponential mechanism, can achieve $\mathcal{O}(1)$ accuracy. The authors of [STU17] note that *general* techniques combining secure computation and differential privacy suffer from bandwidth and liveness constraints, rendering them impractical for large data sets. Our protocol shows that *specially crafted* protocols, combining different techniques and optimizations, achieve performance numbers suitable for practical applications.

2.8 Conclusions

We presented a protocol for secure differentially private median computation on private data sets from two parties with a runtime sublinear in the size of the data universe. Our protocol implements the exponential mechanism as in the local model using a distributed, secure computation protocol to achieve accuracy as in the central model without trusting a third party. For the median the exponential mechanism provides the best utility vs. privacy trade-off for low ε compared to additive noise (see Section 2.2). The output is selected with an exponential bias towards elements close to the median while providing differential privacy for the individuals contained in the sensitive data. We note that our protocol can be easily extended to compute differentially private

rank-based statistics such as p^{th}-percentile and interquartile range. Our experiments evaluate real-world delay and bandwidth, unlike related work [PL15], which we still outperform by at least a factor of 13 (with 25 ms delay and less powerful hardware) by utilizing secret sharing as well as garbled circuits for their respective advantages. We optimize our protocol by computing as little as possible using cryptographic protocols and by applying dynamic programming with a static, i.e., data-independent, access pattern, yielding lower complexity of the secure computation circuit. Our comprehensive evaluation with a large real-world payment data set [Cen17] achieves the same high accuracy as in the central model and a practical runtime of less than 7 seconds for millions of records in real-world WANs. Part of the results obtained by MOSAICrOWN illustrated in this chapter have been published in [BK20].

3. Conclusions

This document reported on the implementation efforts associated with two novel techniques developed for data sanitization in WP5. The two contributions specifically focus on scenarios characterized by multiple interacting parties aimed at anonymize data or at computing privacy-preserving statistics. The application of the protection can be driven by the specification expressed in the MOSAICrOWN policy model, as it was already proposed in Deliverable D3.3 [DS20] and will be expanded in next Deliverable D3.5.

Chapter 1 presents a distributed version of the Mondrian algorithm, aimed at anonymizing large datasets leveraging the presence of several workers that can collaboratively compute a k-anonymous and ℓ-diverse version of the original data collection. The proposed approach limits the interaction among workers by properly partitioning the dataset in such a way to reduce data exchanges and hence improve performance. The chapter also presents the developed tool, implementing our distributed version of the Mondrian algorithm. The experimental results over a sample of the IPUMS USA dataset demonstrate the scalability of the proposed solution, with negligible impact on information loss.

Chapter 2 presents a protocol for computing the median of private data collections owned by two independent parties, while guaranteeing differential privacy. The proposed protocol is flexible and can be extended to support the computation, in a differentially private manner, of other rank-based statistics. The proposed solution is based on the implementation of a secure computational protocol that operates in sublinear time in the size of the data universe. The experimental results demonstrate that the proposed approach outperforms existing approaches, thanks to the adoption of secret sharing and garbled circuits.

Bibliography

[ABPP16] D.W. Archer, D. Bogdanov, B. Pinkas, and P. Pullonen. Maturity and performance of programmable secure computation. In *Proc. of IEEE S&P*, San Jose, CA, USA, May 2016.

[AKS21] F. Ashkouti, K. Khamforoosh, and A. Sheikhahmadi. DI-Mondrian: Distributed improved Mondrian for satisfaction of the ℓ-diversity privacy model using Apache Spark. *Information Sciences*, 546:1–24, 2021.

[AMP10] G. Aggarwal, N. Mishra, and B. Pinkas. Secure computation of the median (and other elements of specified ranks). *Journal of cryptology*, 23:373–401, 2010.

[BDB16] J. Blocki, A. Datta, and J. Bonneau. Differentially private password frequency lists. In *Proc. of NDSS*, San Diego, CA, USA, February 2016.

[BEM+17] A. Bittau, U. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnes, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proc. of SOSP*, Shanghai, China, October 2017.

[BHR12] M. Bellare, V.T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proc. of ACM CCS*, Raleigh, NC, USA, October 2012.

[BIP18] E. Boyle, Y. Ishai, and A. Polychroniadou. Limits of practical sublinear secure computation. In *Proc. of CRYTPO*, Santa Barbara, CA, USA, August 2018.

[BK20] J. Böhler and F. Kerschbaum. Secure sublinear time differentially private median computation. In *Proc. of NDSS*, San Diego, CA, USA, February 2020.

[BSRW17] A.J. Biega, R. Saha Roy, and G. Weikum. Privacy through solidarity: A user-utility-preserving framework to counter profiling. In *Proc. of ACM SIGIR*, 2017.

[CDFS07] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. k-Anonymity. In T. Yu and S. Jajodia, editors, *Secure Data Management in Decentralized Systems*. Springer-Verlag, 2007.

[Cen17] Centers for Medicare & Medicaid Services. Complete 2017 program year open payments dataset. `https://www.cms.gov/OpenPayments/Explore-the-Data/Dataset-Downloads.html`, 2017.

[CSU+19] A. Cheu, A. Smith, J. Ullman, D. Zeber, and M. Zhilyaev. Distributed differential privacy via shuffling. In *Proc. of EUROCRYPT*, 2019.

[DDL78] R.A. DeMillo, D. Dobkin, and R.J. Lipton. Even data bases that lie can be compromised. *IEEE TSE*, 1978.

[DFF+21a] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Artifact: Scalable distributed data anonymization. In *Proc. PerCom 2021*, March 2021.

[DFF+21b] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Scalable distributed data anonymization. In *Proc. of PerCom 2021*, March 2021.

[DFLS12] S. De Capitani di Vimercati, S. Foresti, G. Livraga, and P. Samarati. Data privacy: Definitions and techniques. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 20(6):793–817, December 2012.

[DKM+06] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *Proc. of EUROCRYPT*, Saint Petersburg, Russia, May–June 2006.

[DKY17] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *Proc. of NIPS*, Long Beach, CA, USA, December 2017.

[DL09] C. Dwork and J. Lei. Differential privacy and robust statistics. In *Proc. of ACM STOC*, 2009.

[DMNS06] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. of TCC*, 2006.

[DR14] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[DS20] S. De Capitani di Vimercat and P. Samarati, editors. *D3.3 - First Version of Policy Specification Language and Model*. Deliverable of MOSAICrOWN, June 2020. https://mosaicrown.eu/wp-content/uploads/2020/09/D3.3.pdf.

[DSZ15] D. Demmler, T. Schneider, and M. Zohner. ABY-A framework for efficient mixed-protocol secure two-party computation. In *Proc. of NDSS*, 2015.

[Dwo06] C. Dwork. Differential privacy. In *Proc. of ICALP*, 2006.

[EGL85] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 1985.

[EKM+14] F. Eigner, A. Kate, M. Maffei, F. Pampaloni, and I. Pryvalov. Differentially private data aggregation with optimal utility. In *Proc. of ACSAC*, 2014.

[EPK14] Ú. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proc. of ACM CCS*, 2014.

[GKM+16] V. Goyal, D. Khurana, I. Mironov, O. Pandey, and A. Sahai. Do distributed differentially-private protocols require oblivious transfer? In *Proc. of LIPIcs*, 2016.

[GLM+10] A. Gupta, K. Ligett, F. McSherry, A. Roth, and K. Talwar. Differentially private combinatorial optimization. In *Proc. of ACM SIAM SODA*, Austin, TX, USA, January 2010.

[Gol09] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2009.

[GX17] S. Goryczka and L. Xiong. A comprehensive comparison of multiparty secure additions with differential privacy. *IEEE TDSC*, 14(5):463–477, 2017.

[HEK12] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Proc. of NDSS*, San Diego, PA, USA, February 2012.

[HKR12] J. Hsu, S. Khanna, and A. Roth. Distributed private heavy hitters. In *Proc. of ICALP*, 2012.

[HLM17] N. Holohan, D.J. Leith, and O. Mason. Optimal differentially private mechanisms for randomised response. *IEEE Transactions on Information Forensics and Security*, 2017.

[HMFS17] X. He, A. Machanavajjhala, C. Flynn, and D. Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *Proc. of ACM CCS*, 2017.

[Kag18] Kaggle.com. Walmart supply chain: Import and shipment. `https://www.kaggle.com/sunilp/walmart-supply-chain-data/data`, 2018.

[KLN+11] S.P. Kasiviswanathan, H.K Lee, K. Nissim, S. Raskhodnikova, and A. Smith. What can we learn privately? *SIAM Journal on Computing*, 2011.

[LDR06] K. LeFevre, D.J. DeWitt, and R. Ramakrishnan. Mondrian multidimensional k-anonymity. In *Proc. of ICDE*, Atlanta, GA, USA, 2006.

[LLSY16] N. Li, M. Lyu, D. Su, and W. Yang. *Differential Privacy: From Theory to Practice*. Synthesis Lectures on Information Security, Privacy, & Trust. Morgan & Claypool, 2016.

[LP09] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, April 2009.

[McS09] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proc. of SIGMOD*, 2009.

[MGK06] A. Machanavajjhala, J. Gehrke, and D. Kifer. ℓ-diversity: Privacy beyond k-anonymity. In *Proc. of ICDE*, Atlanta, GA, USA, April 2006.

[MM] D. Mazieres and D. Miller. Source of arc4random.c. `https://opensource.apple.com/source/Libc/Libc-1158.50.2/gen/FreeBSD/arc4random.c`.

[MMP+10] A. McGregor, I. Mironov, T. Pitassi, O. Reingold, K. Talwar, and S. Vadhan. The limits of two-party differential privacy. In *Proc. of IEEE FOCS*, Las Vegas, NV, USA, October 2010.

[MPRV09] I. Mironov, O. Pandey, O. Reingold, and S. Vadhan. Computational differential privacy. In *Proc. of CRYTPO*, Santa Barbara, CA, USA, August 2009.

[MT07] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proc. of IEEE FOCS*, Providence, RI, USA, October 2007.

[NPR19] M. Naor, B. Pinkas, and E. Ronen. How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior. In *Proc. of ACM CCS*, 2019.

[NRS07] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *Proc. of STOC*, 2007.

[PBS12] P. Pullonen, D. Bogdanov, and T. Schneider. The design and implementation of a two-party protocol suite for sharemind 3. Technical report, CYBERNETICA Institute of Information Security, 2012.

[PL15] M. Pettai and P. Laud. Combining differential privacy and secure multiparty computation. In *Proc. of ASAC*, 2015.

[PP02] C.P. Pfleeger and S.L. Pfleeger. *Security in computing*. Prentice Hall, 2002.

[Rab81] M. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard University: Aiken Computation Laboratory, 1981.

[Rep] `https://github.com/mosaicrown/mondrian`.

[RFG+20] S. Ruggles, S. Flood, R. Goeken, J. Grover, E. Meyer, J. Pacas, and M. Sobek. IPUMS USA: Version 10.0 [dataset], 2020. https://doi.org/10.18128/D010.V10.0.

[RN10] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proc. of ACM SIGMOD*, Portland, OR, USA, June 2010.

[Sam01] P. Samarati. Protecting respondents' identities in microdata release. *IEEE TKDE*, 13(6):1010–1027, November/December 2001.

[Soo18] G. Sood. California Public Salaries Data, 2018.

[STU17] A. Smith, A. Thakurta, and J. Upadhyay. Is interaction necessary for distributed private learning? In *Proc. of IEEE S&P*, San Jose, CA, USA, May 2017.

[SV15] A. Shelat and M. Venkitasubramaniam. Secure computation from millionaire. In *Proc. of ASIACRYPT*, Auckland, New Zealand, November–December 2015.

[Tea17] Apple's Differential Privacy Team. Learning with privacy at scale, 2017.

[TKZ16] H. Takabi, S. Koppikar, and S.T. Zargar. Differentially private distributed data analysis. In *Proc. of IEEE CIC*, Pittsburgh, PA, USA, November 2016.

[ULB18] Machine Learning Group ULB. Credit card fraud detection. https://www.kaggle.com/mlg-ulb/creditcardfraud/data, 2018.

[WWD16] WWDC 2016. Engineering privacy for your users. `https://developer.apple.com/videos/play/wwdc2016/709/`, 2016.

[XWP+06] J. Xu, W. Wang, J. Pei, X. Wang, B. Shi, and A.W.-C. Fu. Utility-based anonymization for privacy preservation with less information loss. *ACM SIGKDD Explorations Newsletter*, 8(2):21–30, 2006.

[Yao86] A.C.-C. Yao. How to generate and exchange secrets. In *Proc. of IEEE FOCS*, Toronto, Canada, October 1986.