Combining Differential Privacy and Secure Multiparty Computation

Martin Pettai
Cybernetica AS / STACC
martin.pettai@cyber.ee

Peeter Laud
Cybernetica AS
peeter.laud@cyber.ee

ABSTRACT

We consider how to perform privacy-preserving analyses on private data from different data providers and containing personal information of many different individuals. We combine differential privacy and secret sharing based secure multiparty computation in the same system to protect the privacy of both the data providers and the individuals. We have implemented a prototype of this combination and have found that the overhead of adding differential privacy to secure multiparty computation is small enough to be usable in practice.

1. INTRODUCTION

Many organizations maintain registries that contain private data on the same individuals. Important insights might be gained by these organizations, or by the society, if the data in these registries could be combined and analyzed. The execution of such combination and analysis brings several kinds of privacy problems with it. One of them is *computational privacy* — one must perform computations on data that must be kept private and there is no single entity that is allowed to see the entire dataset on which the analysis is run. Another issue is *output privacy* — it is not *a priori* clear whether the analysis results contain sensitive information traceable back to particular individuals. Kamm [22, Sec. 6.7.1] has presented evidence that the second kind of issues is no less serious than the first kind — even after the computational privacy of data in a study was ensured, one of the data providers (the tax office) was worried about the leaks through the results of the study.

Secure Multiparty Computation (SMC) [37, 17] is a possible method for ensuring the computational privacy of a study. It allows the entity executing the study to be replaced with several entities that perform the computations in distributed manner, while each of them alone or in small coalitions remains oblivious to the input data and the intermediate results. To achieve output privacy, the analysis mechanism itself must be designed with privacy in mind [28].

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA

ISBN 978-1-4503-3682-6/15/12...$15.00

DOI: http://dx.doi.org/10.1145/2818000.2818027

A commonly targeted privacy property is differential privacy (DP) [12, 31], which has both well-understood properties [20] and supports simple arguments, due to its composability.

A statistical analysis mechanism can be designed from ground up with differential privacy in mind. Alternatively, a mechanism, treated as a black-box functionality, can be modified to make it differentially private, albeit with some loss of accuracy. Laplace and exponential mechanisms are basic tools to add differential privacy to a sufficiently smooth mechanism [12]. For many statistical functions, other mechanisms may provide better accuracy for the same level of privacy. In this paper, we will consider the *sample-and-aggregate* method [32], smoothening the function, such that less noise has to be added in order to obtain the same level of privacy.

Differential privacy introduces a generic mechanism for responding to queries about a database containing private data. The mechanism for answering each query has a certain level of privacy loss associated with it; for different queries the levels may be different. There is an overall privacy budget associated with the database. Each given response lowers the available budget by the amount of privacy loss of the mechanism used to construct it. A query is accepted only if its privacy loss does not exceed the remaining budget.

Differential privacy has been generalized to take into account that different records or columns of a database may have different sensitivity [8]. As an instance of generalized definitions, *personalized differential privacy* (PDP) [14] assigns a privacy budget not to the entire database, but to each record separately. The responses to queries may depend only on a subset of all records, and only the budgets of these records will be lowered. This gives more freedom to the data analyst in formulating the queries.

In real applications and database systems, we would like to use both SMC and DP, in order to achieve both computational and output privacy, and to protect the interests of both data owners and data subjects. Alone, both of them have been demonstrated to work with reasonable efficiency, being applicable for realistically sized databases.

In this paper, **our main contribution** is to show that fast methods for SMC and precise methods for DP can be combined, and still provide reasonable performance. We report on the experience that we have obtained with the implementations of GUPT's sample-and-aggregate method [31] and the provenance for PDP method [14] on top of the Sharemind SMC framework. We have implemented a number of statistical functions in this framework, and compared

their performance with and without DP mechanisms. Our results show that the extra overhead of implementing DP mechanisms on top of SMC is not prohibitive.

While implementing the DP mechanisms on top of the SMC framework, we had to come up with novel SMC protocols for some subtasks. These subtasks are related to reading the elements of an array according to private indices; there are no cheap SMC protocols for that and hence the algorithms with a lot of data-dependent accesses incur very significant overhead when straightforwardly converted to run on top of an SMC framework. The subtasks with novel protocols are the following:

- Inner join of two tables. This operation is needed when tracking the provenance of records in queries complying with PDP.
- Counting the number of equal values in an array. This operation is needed for updating the privacy budgets of records. Also, a novel form of updating the elements of an array (where the writing locations are private) is needed for writing back the updated budgets.

The Sharemind SMC framework [4, 5] that we are using employs protocols working on data secret-shared between several computing parties. This gives our databases and queries the greatest flexibility, as we do not have to restrict how the database is stored, and who can make the queries. Indeed, we can just state that the database is secret-shared between the computing parties, and so are the parameters of the query. The answer will be similarly secret-shared. In an actual deployment, the entity making the query will share it among the computing parties. The shares of the answer will be sent back to it, to be recombined. Alternatively, the answer to a query may be an input to further secure computations. In this case it will not be recombined and no party will ever learn it.

2. RELATED WORK

PINQ [29] is one of the best-known implementations of differentially private queries against sensitive datasets. It provides an API for performing such queries, maintaining and updating the privacy budget for a queryable data source.

Rmind [3] is a tool for statistical analysis, preserving computational privacy. It implements a number of statistical operations and functionalities on top of Sharemind, including quantiles, covariance, detection of outliers, various statistical tests and linear regression. The implementations are packaged as a tool resembling the statistics package R.

The combination of SMC and DP has been explored in PrivaDA [15]. They consider the problem of releasing aggregated data collected from many sources. The aggregation (only addition of individual values is considered) is performed using SMC. Afterwards, a perturbation mechanism providing DP is applied to the aggregated data and the perturbed data is made public.

Differentially private data aggregation has received a lot of interest [13, 1, 19], but typically without employing SMC. In this case, each data source itself has to add some noise to its outputs, which will be summed during aggregation. Typically, the noise level in the aggregated result will be larger, compared to adding the noise after aggregation.

Privacy-preserving joins in databases have been considered before in [26]. They obliviously apply a pseudo-random permutation on the key columns of both joined tables, and declassify the results, after which the actual join can be performed in public. Their approach leaks the counts of records that have equal keys.

The reading and updating of arrays according to private indices has been considered before, mostly by implementing the techniques of Oblivious RAM [18] on top of SMC [10, 27]. Our methods have similarities to parallel oblivious array access [25], but differently from them, we do not have to perform computations on stored values.

In our implementation, we had to choose which aggregation functions to implement for queries. We concentrated on count, arithmetic average, median, and linear correlation coefficient. Count has been considered in [12, 29], median in [32, 29]. Arithmetic average has been considered in [29] and is used as the final step in the Sample-and-Aggregate algorithm in [31]. Linear correlation coefficient we chose as an example of a multivariable aggregation function. We have also implemented sum, which is considered in [1], but we do not consider it further in this paper because it is similar to the average.

An alternative to the Sample-and-Aggregate mechanism is the exponential mechanism [30], as an example of which we have implemented a differentially private quantile computation algorithm from [36]. We have not optimized it and thus we do not consider it further in this paper.

3. DIFFERENTIAL PRIVACY

Definition 1. A (probabilistic) algorithm that takes a set of data records as an input, is *ϵ-differentially private* if the removal of any single record from the input changes the probability of any predicate on the output being true at most by a factor of e^{ϵ}.

To achieve differential privacy, one usually adds random noise to the computed result. This noise must have high enough variance to mask the possible variation in the output due to removing or changing one record. This noise is usually from a Laplace distribution, which fits perfectly to the statement of Def. 1.

An alternative would be to use noise from the uniform distribution. This would not satisfy Def. 1 but would instead satisfy the following property:

Definition 2. A (probabilistic) algorithm that takes a set of data records as an input, is *additively δ-differentially private* if the removal of any single record from the input changes the probability of any predicate on the output being true at most by δ.

This has the advantage that the amount of noise is bounded and the average absolute deviation is half of that needed for the Laplace noise, to achieve the same level of indistinguishability for the adversary (a probability of $\frac{1}{2}$ may change to at most $\frac{1}{2} + \delta$ by changing one input record). The disadvantage is that a probability of 0 may change to δ by changing one record, which is not possible with Laplace noise. Thus in the following, we will consider only Laplace noise and Def. 1, which is the mainstream practice.

Suppose we have an algorithm for computing a function f, whose input is a set of data records. To determine how much noise must be added we need to know the sensitivity of f, i.e. how much the value of f can change if we remove one record from its input.

Definition 3. The *sensitivity* of a function $f : 2^{\mathbf{Records}} \to \mathbb{R}$ is

$$\max_{T \subseteq \mathbf{Records}, r \in T} |f(T) - f(T \setminus \{r\})|$$

If the sensitivity of the function is s then adding noise from the distribution Laplace($\frac{s}{\epsilon}$) (this has the average absolute deviation $\frac{s}{\epsilon}$ and standard deviation $\frac{s\sqrt{2}}{\epsilon}$) to the value of f guarantees ϵ-differential privacy.

For example, if f is the arithmetic mean of n values from the range $[a, b]$ then its sensitivity is $\frac{b-a}{n}$. Here it is important that the inputs of f are bounded, i.e. in the range $[a, b]$ for some a and b, otherwise the sensitivity of f would also be unbounded.

In practical analysis, the input values (e.g. salaries) may be from a very wide range. Because the data is private, we may not know the maximum and the minimum of the values. Revealing the exact minimum and the maximum would breach the privacy of the individuals who have those values. Thus we need to guess some values a and b and then clip the input values to the range $[a, b]$ (replacing values smaller than a with a, and values larger than b with b). The range $[a, b]$ must be chosen carefully. If it is too wide, then the added noise (which is proportional to $b - a$) distorts the result too much. If it is too narrow, then the clipping of inputs distorts the result too much.

If a and b differ by orders of magnitude and the distribution of the values is asymmetric in this range (e.g. it may be lognormal) then it may be useful to take logarithms of the values before clipping and later (after adding noise) take the exponent of the final result. Instead of logarithm we may also use other transformation functions, e.g. square root, to make the distribution more symmetric.

If several queries are made where the ith query is ϵ_i-differentially private then the composition of the queries is $(\sum \epsilon_i)$-differentially private. We can define a *(global) privacy budget* B and require $\sum \epsilon_i \leq B$. Thus every query consumes a part of the privacy budget and when a query has a higher ϵ than the amount of budget remaining then this query cannot be executed or the accuracy will be reduced.

4. THE SAMPLE-AND-AGGREGATE MECHANISM

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

A robust method for differentially privately computing the function f is the Sample-and-Aggregate mechanism proposed and investigated by Nissim et al. [32] and Smith [36], and further refined in the GUPT framework [31]. The basic mechanism is given in Alg. 1. Beside the dataset T and the privacy parameter ϵ, Alg. 1 receives as an input a subroutine for computing the function f (without privacy considerations). This subroutine is called by Alg. 1 ℓ times in a black-box manner.

Algorithm 1 The Sample-and-Aggregate algorithm [31]

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

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

Alg. 1 is clearly differentially private due to the noise added at the end. At the same time, Smith [36] shows that if f is *generically asymptotically normal*, then the output distribution of Alg. 1, and the output distribution of f on the same dataset T converge to the same distribution as the size n of T grows (and ℓ grows with it). The convergence holds even if the output dimensionality and clipping range of f, as well as $1/\varepsilon$ grow together with n, as long as the growth is at most polynomial. A statistic is generically asymptotically normal, if its moments are sufficiently bounded; we refer to [36] for the precise definition.

As an example of the Sample-and-Aggregate algorithm, we can compute differentially privately the linear correlation coefficient. The black box in this case takes a dataset as input and computes the non-differentially private linear correlation coefficient of the dataset. We can compute other functions differentially privately by just replacing the black box.

The optimal value of ℓ (giving the best accuracy for the same privacy) may be different for different functions f. [31] suggests using a non-private dataset with a similar distribution to select the optimal value of ℓ. As such non-private datasets are not always available, they take $\ell = n^{0.4}$ by default. [36] proves a convergence theorem for the case $\ell = n^{0.5-\eta}$ where $\eta > 0$ is a small constant (e.g. $\eta = 0.1$).

For computing the arithmetic mean, we can take $\ell = n$ and use one-element blocks with the identity function as the black box. This gives the highest accuracy because the amount of added noise is inversely proportional to ℓ. In our implementation, we use this differentially private arithmetic mean as a subroutine of the Sample-and-Aggregate algorithm.

Another parameter that affects the accuracy is the clipping range $[\mathsf{left}, \mathsf{right}]$. [31] suggests using some of the privacy budget for computing the differentially private α- and $(1 - \alpha)$-quantile of $\{O_i\}$ (by default $\alpha = 0.25$) and using those as the clipping range. As the goal of this paper was to add SMC to differential privacy, and accuracy does not depend much on whether we use SMC or not, we will not discuss accuracy further in this paper.

The privacy parameter ϵ affects both accuracy and privacy. [20] discusses how to choose ϵ for some practical applications (statistical surveys). They assume that there is an upper bound E on the expected cost caused to an individual by disclosing the output of the survey even if he does not participate in the survey. Then ϵ-privacy guarantees that participating does not increase those costs by more than a factor of e^ϵ. Thus each participant must be paid $(e^\epsilon - 1)E$ and the budget sets an upper limit on ϵ.

For computing the median, we could use the Sample-and-

Aggregate algorithm, with the black box returning ℓ elements of the dataset, close to the median. In our implementation, we improve on this by skipping the black box and just taking the ℓ or $\ell+1$ (depending on the parities of ℓ and n) elements *closest* to the median (i.e. the elements on positions $\lceil \frac{n+1-\ell}{2} \rceil$ to $\lfloor \frac{n+1+\ell}{2} \rfloor$ (1-based) in the sorted order).

5. PERSONALIZED DIFFERENTIAL PRIVACY

We have also implemented a mechanism for Personalized Differential Privacy [14]. This uses a more general form of Def. 1, which we give in Def. 4.

Definition 4. Let $\mathcal{E} : \textbf{Records} \to \mathbb{R}$. A (probabilistic) algorithm that takes a set of data records as an input, is *$\mathcal{E}$-differentially private* if the removal of any single record r from the input changes the probability of any predicate on the output being true at most by a factor of $e^{\mathcal{E}(r)}$.

Thus a query can provide a different level of privacy for each record. If $\mathcal{E}(r) = \epsilon$ for all r then we get ϵ-differential privacy as a special case.

We can consider two different methods for enforcing Personalized Differential Privacy. In the simpler case, this means that instead of the global privacy budget, each row in the database has a separate privacy budget. When an ϵ-differentially private query is made, then only the rows participating in the query have their budgets reduced (by ϵ). Thus we have $\mathcal{E}$-differential privacy with $\mathcal{E}(r) = \epsilon$ if the row r participates in the query and $\mathcal{E}(r) = 0$ otherwise. This allows performing more queries using the same privacy budget.

In the more complicated case, each row in the database has a *provenance*, each provenance (not each row) has a privacy budget, and there can be several rows with the same provenance. Thus if an ϵ-differentially private query uses r rows with some provenance p then the budget of this provenance p is reduced by $r\epsilon$. Here we have $\mathcal{E}(p) = r\epsilon$. Here the domain of $\mathcal{E}$ is the set of provenances, not the set of actual records but, as in [14], we can instead consider provenances themselves to be records and take the composed query $Q \circ F$ where F is a union-preserving function that maps each provenance to a set of records with that provenance and Q is the actual query on the chosen records. If Q is ϵ-differentially private then $Q \circ F$ is $\mathcal{E}$-differentially private with $\mathcal{E}(p) = |F(\{p\})| \cdot \epsilon$.

6. SECURE MULTIPARTY COMPUTATION

Secure multiparty computation (SMC) is the universal cryptographic functionality, allowing any function to be computed obliviously by a group of mutually distrustful parties. There exist a number of different techniques for SMC, including garbled circuits [37] and homomorphic encryption [9]. Some practical applications of SMC are considered in [21, 6]. In this work, we have considered SMC based on secret sharing.

In secret sharing there are n parties ($n > 1$), and every private value x is split into shares $x_1, \ldots, x_n$ such that party i has the share x_i. The private value can be recovered if at least k parties out of n provide their shares. For a number of operations, there exist more or less efficient protocols that receive the shares of the operands as input, and deliver the shares of the result of the operation as output to the parties. A number of different protocol sets exist, including the GMW protocol [17], protocols for data shared using Shamir's secret sharing [35, 16], SHAREMIND's protocol set [5] or protocols based on predistributed correlated randomness [2, 11]. The protocols are composable, meaning that they can be combined to solve large computational tasks in privacy-preserving manner. Also, they are *input private*, meaning that no party or a tolerated coalition of parties learns anything new during these protocols, except for the final output.

The framework underlying our implementation is SHAREMIND, which provides protocols for three parties, and is secure against a passive adversary that corrupts at most one party. Compared to other frameworks, it offers protocols for a large set of operations over integers, fixed- and floating-point values, thereby simplifying our implementations and comparisons. The offered protocols are efficient compared to other frameworks, but performing computations on secret-shared private data is still considerably slower than performing the same computations on public data. The difference is especially large for floating-point operations. Thus it is often better to convert private floating-point data to fixed-point data, which can be simulated using integers, and is much faster. When implementing this, we must be careful to avoid overflows. This is especially important for differentially private computations, because an overflow can change the result so much that no reasonable amount of added noise can mask this. Similarly, we must avoid exceptions, e.g. division by zero, since these cannot be masked by noise. Instead, it is necessary to remove the exceptional or overflown values or replace them with default values. This will change the result by a very small amount, because we use ϵ-differentially private algorithms.

Even if we can perform most operations using fixed-point arithmetic, we may still perform a constant number of floating-point operations, e.g. the division when computing an average or the generation of a Laplace random value. For the latter, we first generate a uniformly random value s in the set $\{-1, 1\}$ and a uniformly random value r in the interval $[0, 1]$. Then $s \ln r$ is a Laplace random value. We first generate r as a fixed-point value (each party generates a uniformly random share), then convert it to floating-point, and then compute the logarithm (for which a protocol exists in SHAREMIND).

Because protocols on secret-shared data have a much better performance when a single operation is applied to a large number of values in parallel rather than sequentially (due to the network latency), we may have to structure our differentially private algorithms differently than in the non-secret-shared case. In the Sample-and-Aggregate algorithm (Alg. 1), the set of n inputs is randomly partitioned into ℓ samples, the black box computing f is applied to each of the samples, and the results are combined. It is easy to run the black box on each sample sequentially, but in the secret-shared context we may need to run many copies of the black box in parallel, which would complicate the realization. The sequential algorithm would also work but it would be slower, especially for small values of n/ℓ. For large n/ℓ, each sample would be large enough to fully take advantage of the parallelizability of vector operations, and the difference in performance would diminish.

When implementing an algorithm in privacy-preserving manner, it is generally not possible to branch on a private

condition, because the control flow of the algorithm is visible to all parties. Instead, we must use evaluate both branches and combine them using oblivious choice. If b is private then **if** b **then** c **else** d must be replaced with $b \cdot c + (1-b) \cdot d$, where the boolean b is used as an integer ($\mathsf{true} = 1$, $\mathsf{false} = 0$). This can be further optimized to $d + b \cdot (c - d)$, which uses only one multiplication instead of two. Three-way oblivious choice **if** b_1 **then** c **else if** b_2 **then** d **else** e where b_1 and b_2 are never true at the same time can then be implemented using two multiplications: $e + b_1 \cdot (c - e) + b_2 \cdot (d - e)$. The three-way oblivious choice is used to implement the computation of each O_i in Alg. 1.

To get better performance in the secret-shared setting, the ℓ invocations of the black box in Alg. 1 are done in parallel in single invocation. The input of the black box in this case is a list of datasets and the output is a list of values of f for each dataset. For example, in differentially private computation of the linear correlation coefficient, the black box takes any number of datasets and computes the non-differentially private linear correlation coefficient of each dataset in parallel. For computing other functions differentially privately, we need to replace only the black box.

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

In most cases, it is not difficult to modify the aggregating function to use a mask vector, but in some cases, there can be complications. For example, for computing the median, we replace half of the values excluded by the filter with a very small value, and the other half with a very large value. This will keep the median roughly (exactly if the number of excluded values is even) the same.

When using mask vectors, filtering will not reduce the size of a table. To improve performance, we can reduce the size of the filtered table using a cut operation. This requires an upper bound k on the number of records. The resulting table will have exactly k records (some of which may still be disabled by the mask vector). If there were more than k records, then some elements will be thrown away (uniformly randomly). This distorts the result of the analysis similarly to the clip operation described above, thus the upper bound must be chosen carefully.

The data used in the analysis is in a secret-shared database. Because the data comes from different providers, it will be in different tables. For making the more complex queries, we may need a database join operation to combine two tables. This can be done on secret-shared data in $O(n \log n)$ time (where n is the total number of records in the tables) provided that only one of the two joined tables may have non-unique values in the column used for joining (actually, the other may also contain non-unique values but in this case, for each set of rows with the same key, only one row, chosen uniformly randomly, will be used in the join; this may be acceptable for some applications).

In order to make a query in the system, one may have to provide a range $[a, b]$ (for clipping initial or intermediate values), a number k (for cutting the number of records in an intermediate table), and a value ϵ. These parameters represent the tradeoff between privacy, accuracy, and performance. If we do not know enough about the private data then we can make some preliminary queries to obtain rough estimates for the parameters. These queries should use as small ϵ as possible, to avoid excessive consumption of the privacy budget.

7. ASYMPTOTIC OVERHEAD OF DIFFERENTIAL PRIVACY

Adding differential privacy to a secret-shared aggregation introduces some overhead. In Alg. 2, we give an algorithm for computing (non-differentially privately) in parallel the correlation coefficients of ℓ datasets.

Here and in the following, we use the notation $[\![x]\!]$ to denote the secret shares of x. Variables without the brackets $[\![]\!]$ are public. Thus $[\![x]\!][\![y]\!]$ is the secret-shared product of x and y obtained by applying the secret-shared multiplication protocol to the shares of x and the shares of y. The vector $[\![\vec{c}]\!]$ has elements $[\![c_i]\!]$, where i is public.

Alg. 2 implements the following formula:

$$c_i = \frac{\sum_j x_{ij} y_{ij}}{\sqrt{(\sum_j x_{ij}^2)(\sum_j y_{ij}^2)}}$$

where the values have been normalized by subtracting the corresponding row averages and the values excluded by the mask vector have been replaced with zeros.

Algorithm 2 Parallelized masked correlation algorithm

Input: Dataset matrices $[\![X]\!]$ and $[\![Y]\!]$ (fixed-point numbers) and mask matrix $[\![M]\!]$ (each ℓ rows by m columns), number of blocks ℓ, number of elements in each block m.
Output: For each row $i \in \{1, \ldots, \ell\}$, the correlation $[\![c_i]\!]$ of the ith row of $[\![X]\!]$ with the ith row of $[\![Y]\!]$.

for each $i \in \{1, \ldots, \ell\}$ (in parallel) **do**
  $[\![r_i]\!] \leftarrow \sum_j [\![M_{ij}]\!]$
  $[\![r'_i]\!] \leftarrow \frac{1}{[\![r_i]\!]}$
  $[\![s_i]\!] \leftarrow [\![r'_i]\!] \cdot \sum_j [\![X_{ij}]\!][\![M_{ij}]\!]$
  $[\![t_i]\!] \leftarrow [\![r'_i]\!] \cdot \sum_j [\![Y_{ij}]\!][\![M_{ij}]\!]$
  for each $j \in \{1, \ldots, m\}$ (in parallel) **do**
    $[\![x_{ij}]\!] \leftarrow ([\![X_{ij}]\!] - [\![s_i]\!]) \cdot [\![M_{ij}]\!]$
    $[\![y_{ij}]\!] \leftarrow ([\![Y_{ij}]\!] - [\![t_i]\!]) \cdot [\![M_{ij}]\!]$
  $[\![a_i]\!] \leftarrow \sum_j [\![x_{ij}]\!][\![y_{ij}]\!]$
  $[\![b_i]\!] \leftarrow \sum_j [\![x_{ij}]\!]^2$
  $[\![d_i]\!] \leftarrow \sum_j [\![y_{ij}]\!]^2$
  $[\![c_i]\!] \leftarrow \frac{[\![a_i]\!]}{\sqrt{[\![b_i]\!][\![d_i]\!]}}$
return $[\![\vec{c}]\!]$

In Alg. 2, we first compute $[\![r_i]\!]$, the number of elements in each row i. This is a local operation, so parallelization is not required. Then we compute the inverses $[\![r'_i]\!]$, which

are used to compute $[\![s_i]\!]$ and $[\![t_i]\!]$, the row averages of $[\![X]\!]$ and $[\![Y]\!]$. We do one inverse and two multiplications for each row, which is faster than doing two divisions. When computing the $[\![s_i]\!]$, we can do all $\ell \cdot m$ multiplications $[\![X_{ij}]\!][\![M_{ij}]\!]$ in parallel. The next five sets of $\ell \cdot m$ multiplications each (for computing $[\![x_{ij}]\!]$, $[\![y_{ij}]\!]$, $[\![a_i]\!]$, $[\![b_i]\!]$, $[\![d_i]\!]$) are handled in the same way. Finally, we compute the $[\![c_i]\!]$ by doing ℓ multiplications, divisions, and square roots. Divisions and square roots are expensive operations, so it is important to do them in parallel, even though we only do ℓ of each, not $\ell \cdot m$. If m is small then the divisions and square roots dominate the computation time. If m gets larger then the $O(\ell \cdot m)$ multiplications begin to dominate.

If we compare the computation of the correlation of ℓ samples in parallel to the computation of the correlation of the whole dataset as a single sample then we see that the number of multiplications is almost the same ($7\ell m + \ell$ vs $7\ell m + 1$). The number of divisions increases from 2 to 2ℓ and that of square roots from 1 to ℓ.

When computing correlation differentially privately, we use Alg. 1 with Alg. 2 as a subroutine that is called only once. In addition to the operations done in the subroutine, the algorithm in Alg. 1 does 2ℓ comparisons, 2ℓ multiplications, 1 division (the operations in the argument of Laplace are public), and one generation of a Laplace random value (which uses one division and one logarithm).

We keep our data (the matrices $[\![X]\!]$ and $[\![Y]\!]$ in Alg. 2) in the database in fixed-point form. The multiplications in Alg. 1 and Alg. 2 are integer multiplications but we need to do $2\ell m + 2\ell$ (or $2\ell m + 2$ in the non-differentially private case) shift rights to avoid overflow.

In addition, we need to convert $[\![r_i]\!]$, $\sum_j [\![X_{ij}]\!][\![M_{ij}]\!]$, $\sum_j [\![Y_{ij}]\!][\![M_{ij}]\!]$, $[\![a_i]\!]$, $[\![b_i]\!]$, $[\![d_i]\!]$ in Alg. 2 and $\sum_i [\![O_i]\!]$ in Alg. 1 from integer or fixed-point form to floating point for a total of $6\ell + 1$ conversions. We also need to do 3ℓ conversions to convert $[\![s_i]\!]$, $[\![t_i]\!]$, and $[\![c_i]\!]$ in Alg. 2 from floating point to integer.

We summarize the number of (non-local) operations in both cases:

operation	non-diff. private	diff. private
int multiplication	$7\ell m + 1$	$7\ell m + 3\ell$
shift right	$2\ell m + 2$	$2\ell m + 2\ell$
float multiplication	2	2ℓ
int to float	6	$6\ell + 1$
float to int	2	3ℓ
division	2	$2\ell + 2$
square root	1	ℓ
comparison	0	2ℓ
logarithm	0	1

As we see, for large m, the multiplications and shift rights dominate the running time, and if $m \to \infty$ then the ratios $\frac{7\ell m+3\ell}{7\ell m+1} \to 1$ and $\frac{2\ell m+2\ell}{2\ell m+2} \to 1$, i.e. the overhead of differential privacy is negligible for large block size m. For small m, the $O(\ell)$ overhead may be important.

8. ALGORITHM FOR JOIN

Sometimes the values needed for performing a query are in more than one table (e.g. in Sec. 9). Then we need to join those tables. In this section, we describe an algorithm (Alg. 3) for this. In the following, we call the columns by which the tables are joined, the provenance columns but actually any columns can be used in this role.

Suppose we have matrices $[\![V]\!]$ (with r_V rows and c_V columns) and $[\![B]\!]$ (with r_B rows and c_B columns). Also assume that different rows in $[\![B]\!]$ have different provenances, and all provenances in $[\![V]\!]$ also occur in $[\![B]\!]$. This assumption holds in Sec. 9. We create a bigger matrix

$$[\![A]\!] = \left[\!\!\left[\begin{pmatrix} V & 0 \\ 0 & B \end{pmatrix} \right]\!\!\right]$$

with $r_V + r_B$ rows and $c_V + c_B$ columns. Then we add to it two extra columns to the left, the first of which contains the provenance of the row of $[\![V]\!]$ or $[\![B]\!]$ contained in this row of $[\![A]\!]$. The second column contains 0 or 1 depending on whether the corresponding row of $[\![A]\!]$ contains values from $[\![V]\!]$ or $[\![B]\!]$, respectively.

Now we sort $[\![A]\!]$ by the first column, breaking ties by the second column. Then rows with the same provenance will appear sequentially in $[\![A]\!]$, with the rows from $[\![V]\!]$ appearing before the rows from $[\![B]\!]$ with the same provenance.

Now we apply the algorithm in Alg. 4 to the last c_B columns of $[\![A]\!]$. This algorithm takes a matrix $[\![M]\!]$ with rows sorted by provenance, and copies the last row of each provenance to the previous rows of the same provenance. The while loop does approximately $\log n$ iterations. After the first iteration, there are up to two copies of a value. After the second, up to 2, then 4, 8, and so on. The algorithm makes approximately $n \log n$ equality tests. This technique can be used also for other tasks where we have a matrix sorted by a certain column (provenance) and we want to do something with each group of rows with the same provenance. We will see an example later (Alg. 7).

If the provenance of each row of $[\![B]\!]$ is different then after applying Alg. 4 to the right part of $[\![A]\!]$, the rows of $[\![A]\!]$ with 0 in the second column contain the join of $[\![V]\!]$ and

Algorithm 3 Joining two secret-shared tables

Input: Matrices $[\![V]\!]$ and $[\![B]\!]$

$[\![\vec{s}]\!] \leftarrow$ the provenance column from $[\![V]\!]$
$[\![\vec{t}]\!] \leftarrow$ the provenance column from $[\![B]\!]$
$[\![A]\!] \leftarrow \left[\!\!\left[\begin{pmatrix} \vec{s} & 0 & V & 0 \\ \vec{t} & 1 & 0 & B \end{pmatrix} \right]\!\!\right]$
$[\![A]\!] \leftarrow$ sort $[\![A]\!]$ by the first (provenance) column, breaking ties by the second column
the last c_B columns of $[\![A]\!] \leftarrow$ propagateValuesBack(the last c_B columns of $[\![A]\!]$, the first column of $[\![A]\!]$) using Alg. 4
randomly shuffle the rows of $[\![A]\!]$
declassify the second column of $[\![A]\!]$
$[\![V']\!] \leftarrow$ the rows corresponding to $[\![V]\!]$ (0 in the second column) from $[\![A]\!]$
return $[\![V']\!]$ without the first two columns

Algorithm 4 Propagating values back

Input: An n by c matrix $[\![M]\!]$, an n-element vector $[\![\vec{p}]\!]$ containing the provenance of each row of $[\![M]\!]$

propagateValuesBack($[\![M]\!]$,$[\![\vec{p}]\!]$):
  $j \leftarrow 1$
  while $j < n$ **do**
    for each $i \in \{1, \ldots, n - j\}$ (in parallel) **do**
      $[\![M_i]\!] \leftarrow$ **if** $[\![p_i]\!] = [\![p_{i+j}]\!]$ **then** $[\![M_{i+j}]\!]$ **else** $[\![M_i]\!]$
    $j \leftarrow j \cdot 2$
  return $[\![M]\!]$

$[\![B]\!]$. We may extract the join using a (linear-time) cut operation. This operation is also described towards the end of Alg. 3, starting from the shuffling of the rows of $[\![A]\!]$. The declassification that follows the shuffle does not increase an adversary's knowledge. Indeed, it produces a randomly ordered vector of r_V zeroes and r_B ones, where both r_V and r_B are public information.

9. IMPLEMENTING PERSONALIZED DIFFERENTIAL PRIVACY

Our implementation supports both cases introduced in Sec. 5 but the overhead is much higher in the second case.

We first consider the simpler case. Here we add to the database table an extra column, where we store the privacy budget of each row. There is another column that contains the mask vector that shows which rows participate in the query. When performing an ϵ-differentially private query, we check that each of the rows participating in the query has at least ϵ left in its privacy budget. The rows that do not have enough budget are silently excluded from the query. The other participating rows have their budgets reduced by ϵ. Then the query is executed, using a modified mask vector, where some rows may have been silently excluded. This algorithm is given in Alg. 5. The overhead here is n comparisons (and n multiplications and n boolean operations, which are much cheaper than comparisons).

Algorithm 5 Personalized Differential Privacy with in-place budgets

Input: The number of rows n in the table, privacy parameter ϵ, an ϵ-differentially private query Q

$[\![\vec{m}]\!] \leftarrow$ the mask column read from the database
$[\![\vec{b}]\!] \leftarrow$ the budget column read from the database
for each $i \in \{1, \ldots, n\}$ (in parallel) **do**
  $[\![a_i]\!] \leftarrow [\![m_i]\!] \wedge [\![b_i]\!] \geq \epsilon$
  $[\![b'_i]\!] \leftarrow [\![b_i]\!] - [\![a_i]\!] \cdot \epsilon$
write $[\![\vec{b'}]\!]$ to the database as the new budget column
$[\![r]\!] \leftarrow$ output of Q with $[\![\vec{a}]\!]$ as the mask vector
return $[\![r]\!]$

Now we consider the more complicated case. Here we have in the database a separate table (the budget table) that contains the privacy budget for each provenance. The table containing the analyzed values (the value table) has an extra column that now contains the provenance of that row instead of the budget. Now the data needed for performing a differentially private query is in two separate tables, thus before the query, we need to join those two tables by the provenance columns, and after the query, we need to extract the updated budgets from the joined table and write them to the budget table. Because there may be r rows with a provenance p, the budget of the provenance p must be at least $r\epsilon$, otherwise all the r rows are silently dropped. If the provenance has enough budget, the budget is reduced by $r\epsilon$. Thus the reduction may be different for different provenances. We use Alg. 6 for this case.

It uses the same elements as the join algorithm (Alg. 3)—the big sorted matrix A, propagating values back, and cut (extracting certain rows of a matrix)—but in a modified way, so we cannot use the join algorithm as a black box. In addition, it computes (Alg. 7, which uses the same technique as Alg. 4) the frequency table (the number of rows with each provenance) to determine how much budget is needed for each provenance and which provenances have enough budget (the vector $[\![\vec{h}]\!]$). Then the booleans in $[\![\vec{h}]\!]$ are propagated back from the rows corresponding to $[\![B]\!]$ to the rows corresponding to $[\![V]\!]$ to find the rows whose provenance has enough budget.

We also need to replace the provenances of the rows excluded from the query by zeros ($[\![\vec{s}]\!] \leftarrow [\![\vec{p_V}]\!] \cdot [\![\vec{m_V}]\!]$). This ensures that we do not include those rows when computing the amount of budget required ($[\![f_i]\!] \cdot \epsilon$) for each provenance.

Algorithm 6 Personalized Differential Privacy with provenances

Input: Privacy parameter ϵ, an ϵ-differentially private query Q

$[\![V]\!] \leftarrow$ the value table read from the database
$[\![B]\!] \leftarrow$ the budget table read from the database
$[\![\vec{p_V}]\!] \leftarrow$ the provenance column from $[\![V]\!]$
$[\![\vec{m_V}]\!] \leftarrow$ the mask column from $[\![V]\!]$
$[\![\vec{s}]\!] \leftarrow [\![\vec{p_V}]\!] \cdot [\![\vec{m_V}]\!]$ (multiply elementwise)
$[\![\vec{t}]\!] \leftarrow$ the provenance column from $[\![B]\!]$
$[\![A]\!] \leftarrow \left[\!\!\left[\begin{pmatrix} \vec{s} & 0 & V & 0 \\ \vec{t} & 1 & 0 & B \end{pmatrix} \right]\!\!\right]$
$[\![A]\!] \leftarrow$ sort $[\![A]\!]$ by the first (provenance) column, breaking ties by the second column
$[\![\vec{m}]\!] \leftarrow$ the mask column from $[\![A]\!]$
$[\![\vec{b}]\!] \leftarrow$ the budget column from $[\![A]\!]$
$[\![\vec{p}]\!] \leftarrow$ the provenance (first) column from $[\![A]\!]$
$[\![\vec{f}]\!] \leftarrow$ the frequency table of $[\![\vec{p}]\!]$ using Alg. 7
$n \leftarrow$ the number of rows in $[\![A]\!]$
for each $i \in \{1, \ldots, n\}$ (in parallel) **do**
  $[\![h_i]\!] \leftarrow [\![b_i]\!] \geq [\![f_i]\!] \cdot \epsilon$
  $[\![b'_i]\!] \leftarrow [\![b_i]\!] - [\![h_i]\!] \cdot [\![f_i]\!] \cdot \epsilon$
$[\![\vec{h}]\!] \leftarrow \mathsf{propagateValuesBack}([\![\vec{h}]\!]$ (as a 1-column matrix), $[\![\vec{p}]\!])$ using Alg. 4
for each $i \in \{1, \ldots, n\}$ (in parallel) **do**
  $[\![a_i]\!] \leftarrow [\![m_i]\!] \wedge [\![h_i]\!]$
randomly shuffle the rows of $[\![A]\!]$
declassify the second column of $[\![A]\!]$
$[\![B']\!] \leftarrow$ the rows corresponding to $[\![B]\!]$ (1 in the second column of $[\![A]\!]$) from $([\![\vec{p}]\!], [\![\vec{b'}]\!])$
write $[\![B']\!]$ to the database as the new budget table
$([\![V']\!], [\![\vec{m'}]\!]) \leftarrow$ the rows corresponding to $[\![V]\!]$ (0 in the second column of $[\![A]\!]$) from $([\![A]\!], [\![\vec{a}]\!])$
$[\![r]\!] \leftarrow$ output of Q on $[\![V']\!]$ with $[\![\vec{m'}]\!]$ as the mask vector
return $[\![r]\!]$

Algorithm 7 Frequency table

Input: A sorted vector $[\![\vec{v}]\!]$ of length n

Output: A vector $[\![\vec{f}]\!]$, where $[\![f_i]\!]$ is the number of values equal to $[\![v_i]\!]$ before the ith position in $[\![\vec{v}]\!]$

initialize $[\![\vec{f}]\!]$ with zeros
$j \leftarrow 1$
while $j < n$ **do**
  for each $i \in \{j+1, \ldots, n\}$ (in parallel) **do**
    $[\![f_i]\!] \leftarrow$ **if** $[\![v_i]\!] = [\![v_{i-j}]\!]$ **then** $[\![f_{i-j}]\!] + j$ **else** $[\![f_i]\!]$
  $j \leftarrow j \cdot 2$
return $[\![\vec{f}]\!]$

Let n_v and n_b be the number of rows in the value table and the budget table, respectively, and $n = n_v + n_b$. Then Alg. 6 uses $O(n \log n)$ comparisons for sorting $[\![A]\!]$ (using quicksort) and at most a total of $2n \log n$ equality checks for computing the frequency table and propagating values back (actually, the comparison results from computing the frequency table can be reused for propagating values back, thus we only need $n \log n$ equality checks instead of $2n \log n$). The rest of the algorithm is linear-time.

If we need to make several queries in a row on the same value table and the same mask vector (but with possibly different aggregation functions) then we can reuse (if we modify Alg. 6 slightly) the results of the $O(n \log n)$ part of the algorithm and need to repeat only the linear-time part for each query. If the next query uses the same value table but a different mask vector then we need to recompute the frequency table and propagating values back. Sorting (the most time-consuming part of Alg. 6) needs to be redone only when the next query uses a different value table.

10. BENCHMARKING RESULTS

In Fig. 1, we give benchmarking results of our implemention for various aggregation functions and for various forms of differential privacy (global budgets, Personalized Differential Privacy with in-place budgets and provenance budgets) and also for a non-differentially private (but still secret-shared) version. We have skipped some of the larger tests whose running time would be predictable from the running times of other (performed) tests. All tests were performed on a cluster of three computers with 48 GB of RAM and a 12-core 3 GHz CPU with Hyper Threading running Linux (kernel v.3.2.0-3-amd64), connected by an Ethernet local area network with link speed of 1 Gbps.

If we compare the non-differentially private and the global-budget version of **count**, the overhead is roughly constant (around 360 ms) independent of n (the number of rows). This overhead is due to the floating-point operations of generating a Laplace random value that is added to the final result.

When comparing the non-differentially private and the global-budget version of **average**, we have in addition to generating a Laplace random value, the overhead of $2n$ comparisons and $3n$ multiplications. For $n = 200000$, the overhead is 3427 ms, of which 2638 ms are comparisons, 368 ms are multiplications, and 395 ms are floating-point operations (mostly for the Laplace random value).

When comparing the non-differentially private and the global-budget version of **correlation**, we see that the overhead depends mostly on ℓ and not much on n because the number of slow floating-point operations is proportional to ℓ. For $\ell = 1000$ the running time is about 8000 ms larger than for $\ell = 100$. This extra overhead is used mostly (7000 ms) for floating-point operations (square root, division, etc.).

When comparing the non-differentially private and the global-budget version of **median**, we see that the differentially private version is often the faster one. This is due to the high variance of the running time of the selection algorithm (for choosing the ith ranked element from a set of n elements), and we made only one run for each of the larger input sizes. The distribution of the running time does not depend on the actual input, which is randomly shuffled before running the selection algorithm. In the differentially private version, we do need to choose roughly ℓ middle-ranked elements instead of only 1 or 2 but the running time of the selection algorithm in this case is only $1 + \frac{\ell}{n}$ times higher on average, and usually ℓ is much less than n.

When comparing the global-budget version of differential privacy with in-place budgets, we see that the extra overhead depends mostly on n, not on the aggregating function. This is because we use the same ϵ-differentially private aggregating functions in both cases but in the latter case we also need to check which rows have enough budget and to reduce the budgets.

Similarly, the extra overhead of the provenance-budgets version of differential privacy compared to in-place budgets depends on n and not on the aggregating function.

We have also measured which operations take most of the running time. For example, for **correlation** with provenance budgets, $n = 200000$, and $\ell = 100$, the total running time was 112300 ms, of which 58168 ms is sorting (mostly comparisons), 32949 ms (of which 19697 ms are equality checks and 11707 ms multiplications) is computing the frequency table and propagating values back (Algorithms 7 and 4 but reusing the results of equality checks instead of computing them twice). As discussed in Sec. 9, if the query is performed on the same value table as the previous query then it is not necessary to redo the sorting and the running time would be 54132 ms. If also the mask vector is the same as in the previous query, we can also leave out Algorithms 7 and 4 and the running time would be 21183 ms, which is 2.5 times slower than with in-place budgets but 5.3 times faster than the full provenance-budgets version.

Now we consider **correlation** with in-place budgets, $n = 2000000$, and $\ell = 1000$. Here the overhead compared to global budgets is about 21000 ms, of which 13237 ms are comparisons, 1943 ms are multiplications, and 4345 ms is spent on writing the new budgets to database (a local operation). This overhead would be almost the same for other aggregating functions instead of **correlation**.

Thus the most important operations for our implementation of differential privacy are integer comparisons, followed by equality checks and multiplications. For larger ratios of $\frac{\ell}{n}$ (and thus smaller n), also floating-point operations are important.

11. DISCUSSION

We have implemented our system on Sharemind which provides security against a passive attacker (but also *privacy* against an active one [34]). Floating point operations of Sharemind are described in [23] and integer operations (which we also used to simulate fixed-point numbers) in [5].

We may wonder what the overheads of differential privacy would have been on an SMC platform that provides security also against active adversaries. One of the most efficient actively secure protocol sets is (the online phase of) SPDZ [24]. They use an expensive offline preprocessing phase, which in the online phase allows multiplying two integers with each party sending only two values to every other party (as opposed to five in Sharemind, which does not use preprocessing). Thus (integer) multiplications would be faster on SPDZ but they are only a small part of our algorithms. In the following, we discuss the expected overheads of our protocols, if implemented on top of the online phase of SPDZ.

More important for us are integer comparisons, which in our tests (using 64-bit integers) took about 6.5 s per mil-

function	num. rows	non-diff. private	budgets:		
			global	in-place	provenance
count	10000	392	759	1124	6096
	20000	405	768	1241	10672
	50000	460	827	1587	
	100000	594	962	2257	
	200000	866	1184	3452	
	500000			7234	
	1000000			13871	
average	10000	443	1099	1475	6598
	20000	461	1285	1753	11051
	50000	532	1774	2531	
	100000	694	2625	3873	
	200000	999	4426	6483	
	500000			15118	
	1000000			28995	
correlation ($\ell = 100$)	10000	822	2455	2826	7663
	20000	1001	2665	3157	12693
	50000	1556	3278	4092	26833
	100000	2473	4312	5525	54767
	200000	4443	6359	8548	112300
	500000	9721	12218	17895	
	1000000	19414	22608	33530	
correlation ($\ell = 1000$)	500000	9850	20115	25961	
	1000000	19380	30657	41363	
	2000000	37643	51381	72436	
median ($\ell = 100$)	10000	851	1332	1708	6548
	20000	1327	1737	2131	11786
	50000	2318	2137	3502	
	100000	4620	4075	5312	
	200000	6498	5559	9198	
	500000			24175	
	1000000			34583	

Figure 1: Benchmarking results (in milliseconds)

lion elements. Multiplications took 0.5 s per million elements and equality tests 3 s per million elements. As SPDZ multiplications use 2.5 times less communication, these may take 0.2 s per million elements on our hardware. According to [24], 64-bit integer comparisons in SPDZ are about 90 times slower than multiplications, i.e. these may take 18 s per million elements, about 3 times slower than on Sharemind. Equality tests are not considered in [24] but the best equality-checking protocol in [7] makes 8 openings of secret values for 64-bit integers in the online phase (and much more in the precomputing phase), i.e. 4 times more than multiplication. If this could be implemented on SPDZ then it may take 0.8 s per million elements, about 4 times faster than on Sharemind. Thus we guess the that the communication costs of the online phase of an implementation on SPDZ would not differ from our implementation by more than a couple of times.

12. CONCLUSION

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

Acknowledgements. This work has been supported by Estonian Research Council, grant No. IUT27-1, by European Regional Development Fund through STACC, and by European Commission through grant No. FP7-284731.

References

[1] G. Ács and C. Castelluccia. I have a dream! (differentially private smart metering). In T. Filler, T. Pevný, S. Craver, and A. D. Ker, editors, *Information Hiding - 13th International Conference, IH 2011, Prague, Czech Republic, May 18-20, 2011, Revised Selected Papers*, volume 6958 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2011.

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

[3] D. Bogdanov, L. Kamm, S. Laur, and V. Sokk. Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512, 2014. http://eprint.iacr.org/.

[4] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In S. Jajodia and J. López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[5] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.

[6] H. Carter, C. Lever, and P. Traynor. Whitewash: outsourcing garbled circuit generation for mobile devices. In Payne Jr. et al. [33], pages 266–275.

[7] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In J. A. Garay and R. D. Prisco, editors, *Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings*, volume 6280 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2010.

[8] K. Chatzikokolakis, M. E. Andrés, N. E. Bordenabe, and C. Palamidessi. Broadening the scope of differential privacy using metrics. In E. D. Cristofaro and M. Wright, editors, *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, volume 7981 of *Lecture Notes in Computer Science*, pages 82–102. Springer, 2013.

[9] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In B. Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

[10] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In Y. Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2011.

[11] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

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

[13] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In S. Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 486–503. Springer, 2006.

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

[15] F. Eigner, M. Maffei, I. Pryvalov, F. Pampaloni, and A. Kate. Differentially private data aggregation with optimal utility. In Payne Jr. et al. [33], pages 316–325.

[16] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography. In *PODC*, pages 101–111, 1998.

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

[18] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[19] S. Goryczka, L. Xiong, and V. S. Sunderam. Secure multiparty aggregation with differential privacy: a comparative study. In G. Guerrini, editor, *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, pages 155–163. ACM, 2013.

[20] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 398–410. IEEE, 2014.

[21] N. Husted, S. Myers, A. Shelat, and P. Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In C. N. Payne Jr., editor, *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 169–178. ACM, 2013.

[22] L. Kamm. *Privacy-preserving statistical analysis using secure multi-party computation*. PhD thesis, University of Tartu, 2015.

[23] L. Kamm and J. Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, pages 1–18, 2014.

[24] M. Keller, P. Scholl, and N. P. Smart. An architecture for practical actively secure MPC with dishonest majority. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 549–560. ACM, 2013.

[25] P. Laud. Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees. *Proceedings on Privacy Enhancing Technologies*, 1, 2015. To appear.

[26] S. Laur, R. Talviste, and J. Willemson. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *Applied Cryptography and Network Security*, volume 7954 of *LNCS*, pages 84–101. Springer, 2013.

[27] C. Liu, Y. Huang, E. Shi, J. Katz, and M. W. Hicks. Automating efficient ram-model secure computation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 623–638. IEEE Computer Society, 2014.

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

[29] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. *Commun. ACM*, 53(9):89–97, 2010.

[30] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*, pages 94–103. IEEE Computer Society, 2007.

[31] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. E. Culler. GUPT: privacy preserving data analysis made easy. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 349–360. ACM, 2012.

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

[33] C. N. Payne Jr., A. Hahn, K. R. B. Butler, and M. Sherr, editors. *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*. ACM, 2014.

[34] M. Pettai and P. Laud. Automatic Proofs of Privacy of Secure Multi-Party Computation Protocols Against Active Adversaries. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 14-17 July, 2015*. IEEE, 2015.

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

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

[37] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.