Differentially Private Intermediate Result Resizing for Scalable Secure Multi-Party Analytics*

Long Gu[1], Gowri R Chandran[2], Shaza Zeiouni[1], Thomas Schneider[1], and Zsolt István[1]

[1] Technical University of Darmstadt, Germany
[2] Simula UiB, Bergen, Norway

Abstract. Secure Multi-Party Computation (MPC) enables collaborative analytics without exposing raw data. Yet, complex data retrieval operations in relational databases (i.e., relational queries) remain limited by scalability bottlenecks: oblivious operators must pad intermediate results to worst-case sizes to prevent information leakage, thereby inflating communication and computation costs. We propose Resizer, a lightweight operator that can be inserted transparently into relational query plans to bound intermediate result sizes while preserving rigorous privacy guarantees. Resizer reduces intermediate results from fully-oblivious to noisy size bounds that satisfy differential privacy (DP). We propose two Resizer variants, shuffle-based and sort-based, and provide a formal analysis of the privacy guarantees and security in the semi-honest model. We evaluate the Resizer variants in the ORQ framework under semi-honest and honest-majority assumptions using TPC-H benchmark queries, demonstrating that Resizer incurs negligible overhead while reducing data volume. In both LAN and WAN settings, complex queries achieve speedups compared with state-of-the-art oblivious baselines, confirming that Resizer reduces the padding bottleneck and enables scalable, privacy-preserving relational analytics.

Keywords: Differential Privacy · Secure Collaborative Analytics · Oblivious Computation · Differential Obliviousness.

1 Introduction

In today's data-driven world, organizations rely heavily on analytical data processing for strategic decision-making. Secure Collaborative Analytics (SCA) enables multiple parties to collaborate on data analysis while maintaining the privacy and security of sensitive information [2,25,26,33,4]. Thus, fostering collaboration without compromising confidentiality. This is particularly crucial in industries such as healthcare, advertising, and finance, including applications like financial fraud detection, where data sensitivity is paramount.

* Please cite the conference version published at ESORICS 2026

In SCA, each party contributes its dataset in a relational form: the collective input is represented as a set of relations (tables), each table consisting of a fixed schema of attributes (columns) and populated by tuples (rows) that hold the actual values. Expressed in this tabular model, the analytical workload can be described by standard relational operations: projection and selection to reshape or filter tables based on predicates (i.e., Boolean conditions over attribute values), joins to combine rows from different tables on common keys (also governed by predicates), and aggregations (e.g., SUM, COUNT, AVG) to compute summary statistics.

Multi-Party Computation (MPC) provides a privacy-preserving approach for SCA that minimizes trust requirements, enabling organizations to jointly derive insights without disclosing their underlying raw data. Private datasets are encoded, e.g., via secret sharing, and relational operations are performed via MPC protocols. Nevertheless, MPC-based SCA faces a significant efficiency challenge. Oblivious execution of a query requires padding intermediate results of operators to fully conceal data-dependent volume and execution characteristics. For this, each operator expands its output to the worst-case cardinality by introducing filler/dummy tuples. While this approach fully prevents information leakage, it induces exponential data expansion—most notably in relational `Join` operations. For `Joins`, the worst-case cardinality is the Cartesian Product of the two input tables, each of size $|R|$ and $|S|$, resulting in $|R \times S|$ rows [25,22]. With multiple `Joins`, data sizes increase rapidly, resulting in substantial performance and efficiency degradation. More details on MPC-based SCA are in § A.

To address this challenge, researchers have identified opportunities to improve SCA's practicality by allowing controlled information leakage via Differential Privacy (DP) [10,3,9,31]. In this setting, intermediate results are padded to a differentially private noisy bound rather than the worst-case size, thereby obscuring exact data volumes while reducing overhead.

1.1 Our Contributions

Novel Resizing/Padding Mechanism. We introduce *Resizer*, a novel mechanism that reduces the number of filler tuples in intermediate results produced by oblivious database operators. Resizer can be transparently inserted into relational query plans, providing a principled means to regulate the growth of filler tuples in intermediate results while preserving the formal guarantees of DP. Given a query plan, a Resizer instance may be placed after each operator to bound the fully-oblivious intermediate result size (IRS) and prevent ballooning of IRS toward the end of the plan. Each Resizer is parameterized by $p_i \in [0, 1]$, the success probability of a biased coin flip that determines how many filler tuples are retained in addition to the true output of the preceding operator. This resembles the database notion of operator selectivity, i.e., the proportion of input tuples that satisfy a predicate of a relational operator. We present the Resizer concept and variants in § 4.

Formal Security and Privacy Proofs. We provide formal security and privacy analyses of the entire resizing pipeline. The core stochastic primitive is a biased coin flip with success probability p_i, which we use to design a Mangat-style randomized response mechanism [27]. In § 5, we prove that this mechanism is differentially private. Moreover, in § 6, we prove that the sequential execution of the Resizer mechanism is secure against a semi-honest adversary and does not leak any additional information about the database.

Privacy Budget Control. We aim to move away from a rigid, predefined global privacy budget that requires prior knowledge of how many times a dataset can be queried before the privacy budget is exhausted. Instead, we propose a bottom-up privacy-budgeting approach in which the privacy-performance of each operator can be tuned individually. By adjusting each Resizer's success probability p_i, we can trade off 1) the amount of noise introduced to preserve privacy and 2) the computational overhead associated with processing larger intermediate results passed to subsequent relational operators. This enables a precise balance between privacy and performance. Resizers' privacy losses are then aggregated along the query plan using standard differential privacy composition theorems to determine the query-level privacy loss. The query-level losses are accumulated across successive queries to monitor the system-wide privacy expenditure. This data-flow-driven accounting is particularly well-suited for dynamic workloads, where the number and shape of queries are unpredictable. We discuss privacy-performance tuning in § 5.1.

Performance Assessment. We extend ORQ[4], a state-of-the-art data-parallel engine for MPC-based SCA, with two new operators: the two Resizer variants. Implemented on a three-party honest-majority protocol, the Resizers are assessed under both WAN and LAN settings. Our experiments show the intrinsic overhead of the Resizer, its effect when placed after common relational primitives (`Filter`, `Join`, `GroupBy`), and its positive impact on end-to-end query performance: the speedups thanks to the substantial reductions in intermediate result sizes outweigh the additional computation cost. In addition to micro-benchmarks, we use three complex TPC-H [36] queries to showcase the benefits of the Resizer.

2 Background

2.1 Multi-Party Computation (MPC)

MPC allows several mutually distrusting parties to compute a joint function on their private inputs without revealing their inputs. MPC protocols can be broadly classified into two categories: Garbled Circuit (GC) based [38,6], and Secret Sharing (SS) based [28,13]. In GC-based protocols, one party generates a garbled circuit of the function being computed using its own inputs, and the other party evaluates the circuit based on its own input. GC-based protocols are low-latency and require a constant number of rounds [23,7,32,39] for the computation. On the other hand, in SS-based protocols, the parties split their private

inputs into secret shares and compute the function on these shares. These protocols are high-throughput; however, they require communication rounds that scale linearly with the depth of the circuit being computed [1,11,30,29]. To address this, several works use a preprocessing model [5,8,14,24], where the computation is split into an input-independent preprocessing phase that can be performed offline before the computation begins. As a result, the cost of online communication is reduced drastically, enabling more efficient protocols.

2.2 Differential Privacy

Differential Privacy (DP) [16] is a technique commonly used in private data analysis to protect individual data privacy. It ensures that the contribution of any single data item has a limited effect on the output, thereby bounding the amount of information an adversary can infer about an individual. The strength of this privacy guarantee is controlled by the privacy parameter ϵ, where smaller values provide stronger protection. DP is formally defined as follows:

Definition 1 ((*Pure*) Differential Privacy [16]). *A mechanism $\mathcal{M}$ is ϵ-differentially private if for every pair of neighbouring datasets $D \sim D' \in X^n$, and for all functions $f \in Q$, we have*

$$\forall z \in Z,\ \Pr[\mathcal{M}(D, f) = z] \leq e^{\epsilon} \cdot \Pr[\mathcal{M}(D', f) = z]. \tag{1}$$

Datasets D and D' are neighbouring (denoted as $D \sim D'$) if they differ on one row. Depending on the application, the definition of DP can be relaxed slightly to allow a negligible statistical distance term, δ, in addition to the multiplicative term, ϵ. This is known as *approximate DP*, and is defined as follows:

Definition 2 (Approximate Differential Privacy [37]). *For $\epsilon \geq 0,\ \delta \in [0, 1]$, we say that a randomized mechanism $\mathcal{M} : X^n \times Q \to Y$ is* (ϵ, δ)-differentially private *if, for every two neighboring datasets $D \sim D' \in X^n$ and every function $f \in Q$, we have*

$$\forall z \in Z,\ \Pr[\mathcal{M}(D, f) = z] \leq e^{\epsilon} \cdot \Pr[\mathcal{M}(D', f) = z] + \delta. \tag{2}$$

DP Composition DP composition allows the construction of complex differentially private algorithms as a composition of simpler differentially private building blocks. The basic DP composition works as follows:

Lemma 1 (Basic DP composition [37]). *If $\mathcal{M}_1, \ldots, \mathcal{M}_k$ are each (ϵ, δ)-differentially private, then $\mathcal{M}$ is $(k\epsilon, k\delta)$-differentially private, where $\mathcal{M}(D) = (\mathcal{M}_1(D), \mathcal{M}_2(D), \cdots, \mathcal{M}_k(D))$.*

A standard DP mechanism adds noise, so both false positives and false negatives appear in the final result. However, to improve the accuracy of query evaluations, we avoid false negatives. This ensures that the true results are always included in the final result. To achieve this, we use a variation of the standard DP mechanism called *asymmetric* DP (ADP) [35], which adds only one-sided noise. To design an ADP mechanism, we first define a policy function q, which forms the basis of the asymmetry.

Definition 3 (Policy q). *q is a policy function $q : \chi \to \{\text{True}, \text{False}\}$ defined as*

$$q(x) = \begin{cases} \text{True} & \text{if } v = 0 \\ \text{False} & \text{if } v = 1 \end{cases}$$

where $v = 1$ indicates a true positive result.

Definition 4 (q-neighbouring). *D and D' are datasets, and q is a policy function (see Definition 3). D is q-neighbouring D' (denoted as $D \sim^q D'$) if $D \sim D'$ and ($\neg q(x)$ or $(q(x) \wedge q(x'))$ =False, for $x \in D,\ x' \in D'$, and $x \neq x'$.*

Now, asymmetric DP (ADP) can be defined as follows:

Definition 5 (Asymmetric DP). *A randomized mechanism M satisfies (ϵ, q)-ADP iff $\forall z \in \mathcal{Z}, D, D'$ such that $D \sim^q D'$*

$$Pr[\mathcal{M}(x) = z] \leq e^{\epsilon} Pr[\mathcal{M}(x') = z].$$

3 System, Adversary and Leakage Models

System Model. SCA is a set of distributed data statistics protocols designed to protect private datasets provided by *Data Owners*. Let $\mathcal{DO} = \{DO_1, \ldots, DO_m\}$ be a set of m *Data Owners*, where each DO_i holds a private database D_i. SCA protocols enable joint data analysis on the union of all private datasets $\mathcal{D} = \{D_1, \ldots, D_m\}$ while protecting the privacy of individual datasets. Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ denote a set of n computing nodes or parties that are responsible for executing SCA protocols on $\mathcal{D}$. The *Data Analyst* sends a query Q to be executed over the union of all private datasets $\mathcal{D}$ to $\mathcal{P}$ and collects the final result R. In the MPC-based outsourcing model of SCA, each data owner DO_i provides the secret shares, determined by the underlying MPC protocol, of the dataset D_i to the computing nodes. After a query completes, the query and its output are discarded. A query is translated into a set of MPC protocols π, which are executed by all computing nodes $\mathcal{P}$ on the secret shares of the union of all datasets $\mathcal{D}$. It is a common assumption that in MPC, all parties, i.e., data owners, computing nodes, and the data analyst, know the computed function, i.e., the query, and the database schema, including relations' sizes, are public.

Adversary Model. In this setting, the data owners faithfully follow the protocol for generating and transmitting the secret shares of 1) their datasets and 2) the resizers' success probabilities (sent for each query). The data analyst is trusted to issue queries deemed acceptable by the data owners and to process the query results correctly, without attempting to learn any additional information about the underlying datasets. Nevertheless, differential privacy can be applied to the query response to protect individual privacy [19]. Queries approval is assumed to be handled by policy-based access controls, which are outside the

scope of this work. Once the privacy budget is exhausted, subsequent queries are executed fully obliviously via MPC.

Under these assumptions, the system is designed to protect the confidentiality of the datasets during computation. We consider a *semi-honest* adversary model, in which computing nodes may be corrupted by an adversary but follow the prescribed protocol correctly. In the normal operating scenario, the nodes are assumed to be non-colluding, and at most t out of n nodes may be corrupted simultaneously. If a node becomes compromised, it may collude with other compromised nodes, sharing all locally observed information (e.g., data volumes, randomness, and exchanged messages) to infer additional details about the input datasets.

Leakage Model. To improve efficiency, strict full obliviousness is traded off for performance by introducing controlled leakage via differential privacy [10]. That is, intermediate results are padded to a differentially private noisy bound rather than the worst-case size, thereby obscuring exact cardinalities while reducing performance overhead. State-of-the-art MPC-based SCA systems [25,33,4] implement oblivious operators that scan the entire relation, ensuring uniform access to all tuples. However, when padding the intermediate results to a noisy size rather than the worst case, the uniformity guarantee is broken: only noisy intermediate results are accessed. After multiple executions of the same predicate, the attacker can recover/approximate true IRS [17].

Beyond IRS leakage, structural leakage may also arise. When the physical or logical layout of tuples in memory remains (partially) stable across executions, the attacker can infer the locations of positive (i.e., matching) tuples. Specifically, tuples that evaluate to false (i.e., true negatives) can be distinguished from noise-induced padding (false positives), which vary across executions. By iteratively identifying false-positive tuples, the adversary can infer the positions of true-positive tuples. Therefore, even when exact cardinalities are obscured, repeated executions of queries on a fixed layout of tuples introduce an additional leakage channel that must be explicitly accounted for, as shown in § 4.

4 Resizer Overview

The Resizer can be placed after any oblivious operator to reduce its output size by selectively discarding filler tuples. It works by evaluating each tuple individually: real (true) tuples that come from the operator's output are always kept, while filler tuples are only retained with a certain probability p, determined by a weighted coin flip.

The Resizer consists of two main components. First, it determines, in two steps: *Coin Toss* and *Mark*, which filler tuples to retain. Second, it removes any excess filler tuples. The removal can be implemented in two ways, yielding two variants of the Resizer: the *shuffle-based* and the *sort-based* Resizer. The shuffle-based variant randomizes the order of tuples after Mark, whereas the sort-based variant reorders tuples based on whether they are to be kept, as in

related work [3] (cf. § 7). After *Shuffle*, respectively *Sort*, unmarked filler tuples are removed before the intermediate result is given as input to the next operator.

Overall, the Resizer follows a sequence of four processing steps: *Coin Toss*, *Mark*, *Shuffle*/*Sort*, and *Trim* (cf. Figure 1). It takes as input a success probability p, the output $\mathcal{O}$ of the preceding query operator padded to worst-case size N, and the validity mask v that indicates whether a tuple is a part of the operator's true output. The Resizer returns the output of the preceding operator padded only to a noisy size S, where $S \leq N$. The Resizer is performed via MPC and reveals no information about its inputs, p, v, and $\mathcal{O}$, apart from the final *trimmed* output size S.

Ideal Functionality f_R

Input: preceding operator's output (relation $\mathcal{O}$, validity mask v), and probability p.
Functionality:
Step 1: **Toss** a p-biased coin N times to get a vector ℓ.
Step 2: For each row j, if $v[j] = 1$, **mark** $k[j] = 1$; else **mark** $k[j] = \ell[j]$.
Step 3: **Shuffle** or **Sort** the rows of relation $\mathcal{O}$ and v to obtain $\acute{\mathcal{O}}$ and $\acute{v}$.
Step 4: **Trim** $\acute{\mathcal{O}}$ and $\acute{v}$ to include only rows with $k[j] = 1$ and obtain $\tilde{\mathcal{O}}$ and $\tilde{v}$.
Output: Database $\tilde{\mathcal{O}}$ with attribute $\tilde{v}$.

Fig. 1: Ideal Functionality f_R for Resizer.

Step 1: Coin Tossing. The *Coin Toss* step takes as input a success probability p. Conceptually, a weighted coin with bias p is flipped independently N times, once per output tuple, to determine whether a tuple is retained. Concretely, for each tuple, an independent random value $rand[j]$ is sampled and compared against p. The result is encoded as a single-bit attribute (column) ℓ, where each bit indicates the outcome of the corresponding coin flip (i.e., whether the coin flip succeeds). The selection of p value is discussed in § 5.1.

Step 2: Marking. The *Mark* step takes as input the outcome of the coin flip ℓ, and the attribute v, which identifies the operator's true tuples. Mark outputs a column k, indicating tuples to keep from trimming. For each tuple j:

- If the tuple is part of the true output (i.e., $v[j] = 1$), its corresponding bit $k[j]$ is set to 1, regardless of the coin toss result.
- Else, i.e., the tuple is not part of the true output ($v[j] = 0$), the outcome of the coin toss determines its inclusion in the output. A successful toss (i.e., $\ell[j] = 1$) sets $k[j] = 1$, while a failure excludes it by setting $k[j] = 0$.

The *Coin Toss* and *Mark* steps can be fused into a single loop and further integrated with the preceding operator, thereby avoiding additional scans over

the data and reducing the overall performance overhead. Furthermore, their tuple-level independence enables natural parallelism, greatly reducing Resizer's performance overhead. This makes the method well-suited for MPC execution, as it requires fewer communication rounds.

Step 3: Shuffling or Sorting. Given that an adversary can recover the true tuples' layout after observing several executions (cf. § 3), the Resizer incorporates a shuffling or a sorting step, depending on the variant used. The purpose of shuffling or sorting is to prevent linkage of a tuple's encoding as it propagates through the query operators or when observations are correlated across multiple queries. As discussed in § 9, the choice of shuffling vs. sorting depends on the properties of the network.

Step 4: Trimming. In *Trim*, the computing nodes reconstruct k and discard tuples with $k[j] = 0$. The output after the Resizer satisfies $S = T + \eta \leq N$, where T is the true output size of the preceding operator. The N times coin tossing is represented with the Binomial distribution, and η is the Binomial distribution's expected value, taking into consideration the true match size: $\eta = p(N - T)$. As discussed in § 8, the implementation of the Trim step includes an additional memory copy of individual tuples after Shuffle.

5 Resizer Privacy

The Coin toss and Mark step of the Resizer together are responsible for introducing randomness into the IRS. In the subsequent Trim step, the fully oblivious IRS is reduced. We show that this coin-tossing and marking mechanism preserves privacy by proving that the released size is differentially private. Consequently, the computing nodes cannot infer the presence or absence of any individual's data in the dataset. Since the marking mechanism depends on the true output of the operator and does not remove any true results, thereby preserving accuracy, we show that the Resizer mechanism satisfies asymmetric differential privacy (ADP) (cf. § 2.2).

First, we formally define how the Resizer mechanism adds the randomness to the size, and then we prove that this mechanism is (ϵ, q)-ADP, where ϵ is the privacy parameter, and q is a policy that defines the asymmetry (cf. Definition 3).

Definition 6 (Resizer Mechanism). *The Resizer Mark step works as follows:*

> *If $v[j] = 1$, set $k[j] = 1$. Else if $v[j] = 0$, flip a coin with bias p; Set $k[j] = 1$ if HEAD, and $k[j] = 0$ if TAIL.*

The size of the database after the trim step can be written as:

$$S = T + \eta,$$

where η is drawn from a binomial distribution. The resizer mechanism behaves similarly to Mangat Randomized Response.

Theorem 1. *The Resizer mechanism (Definition 6) is* (ϵ, q)*-ADP.*

Proof. Consider datasets D and D' such that $D \sim^q D'$. Let V be the set of true outputs of the operator *op*, i.e., $V = \{x_j | v[j] = 1\}$. From the policy function q, $q(x) = True \implies x \notin V$. Let $x' \in D'$, such that $q(x') = False \implies x' \in V$. Then, for all operators *op* with sensitivity 1,

$$T = T' - 1.$$

Performing the Resizer Mechanism on these datasets, we get

$$\frac{Pr[M(D) = S]}{Pr[M(D') = S]} = \frac{Pr[M(D) = T + \eta]}{Pr[M(D') = T' + \eta']} = \frac{Pr[Z = \eta]}{Pr[Z = \eta - 1]} \tag{3}$$

$$= \frac{\binom{N}{\eta} \beta^{\eta} (1-\beta)^{(N-\eta)}}{\binom{N}{\eta - 1} \beta^{\eta - 1} (1-\beta)^{(N-(\eta-1))}} \tag{4}$$

$$= \frac{N - \eta + 1}{\eta} \cdot \frac{\beta}{1-\beta}, \tag{5}$$

where $\beta = \gamma + (1-\gamma)p$ is the probability that $k[j] = 1$, and $\gamma = T/N$ is the proportion of true results. Solving the function $R(\eta) = \dfrac{N - \eta + 1}{\eta} \cdot \dfrac{\beta}{1-\beta}$ for $\eta = 1$, $R_{max} = N \cdot \dfrac{\beta}{1-\beta}$, and for $\eta = N$, $R_{min} = \dfrac{1}{N} \cdot \dfrac{\beta}{1-\beta}$.
Then, the mechanism satisfies differential privacy when

$$\epsilon \geq \ln\left(\max\left(R_{max}, 1/R_{min}\right)\right) = \ln\left(N \cdot \max\left(\frac{\beta}{1-\beta}, \frac{1-\beta}{\beta}\right)\right). \tag{6}$$

Stronger privacy is obtained when ϵ is minimized; this occurs at $\beta = 1/2$, where $\epsilon = \ln N$. To examine the remaining cases: consider when $\beta < 1/2$; in this case, $1 - \beta/\beta > 1$, and $\epsilon = \ln\left(N \cdot \dfrac{1-\beta}{\beta}\right)$. Now, consider when $\beta > 1/2$; then, $\beta/1 - \beta > 1$, and $\epsilon = \ln\left(N \cdot \dfrac{\beta}{1-\beta}\right)$.

Therefore, under the conditions outlined above, the IRS produced by the application of the Resizer satisfies asymmetric differential privacy (ADP).

The above proof is valid for all SQL operators having a sensitivity of $s = 1$. For the operators with sensitivity other than $s \neq 1$, for instance `Join` or `GroupBy`, the privacy guarantees follow a generalization of Equation 5. The generalized equation and detailed analysis of the conditions are discussed in § B.

5.1 Selection of parameters

One of the key highlights of our Resizer mechanism is its ability to balance privacy and performance through its configurable success probability p, which

directly affects both the privacy loss ϵ and performance via the number of filler tuples $p(N-T)$ padding the intermediate results. By adjusting p, the users gain fine-grained control over the trade-off between privacy and performance. Once the privacy budget is assigned per operator, the total query privacy budget can be computed as the composition of differentially private mechanisms (cf. Lemma 1), yielding a linear composition of the privacy losses of each Resizer operator in the query plan. This is in contrast to a per-query budget distribution, which is commonly adopted in prior work [3,31,12]. Our approach makes budget allocation easier, as users do not need to know the exact number of queries in advance. We discuss next how p selection affects the protocol's privacy and performance, and how to select them for optimal conditions.

Tuning for Higher Privacy. The best level of privacy is achieved when the intermediate results are fully oblivious, i.e., padded to the worst-case size. However, this significantly reduces the performance. Therefore, the objective of privacy tuning is to achieve a high level of privacy while improving performance over fully oblivious execution. With our proposed Resizer, the best privacy is achieved when the privacy parameter $\epsilon = \ln N$. This privacy level is achieved when $\beta = Pr[k[j] = 1] = 0.5$. This can only be achieved for operators with low selectivity, i.e., $\gamma \leq 0.5$. Then, the Resizer's success probability (the coin toss's bias p) can be chosen accordingly. For an operator with selectivity $\gamma > 0.5$, optimal privacy can be achieved by selecting a small $p \gtrsim 0$.

Tuning for Higher Performance. To improve the performance of MPC-based SCA, intermediate results are only padded to a noisy bound defined by the Coin Toss and Mark steps. To increase efficiency, the number of filler tuples should be kept to a minimum. i.e., this is achieved by setting the probability $p \to 0 \implies \beta \to \gamma$, while maintaining differential privacy, as in Equation 6. In this case, privacy depends on the true IRS, and it is best when the true IRS equals $N/2$ (i.e., $\gamma \to 1/2$).

6 Resizer Security

In this section, we prove the security of the Resizer, which consists of four sequentially executed steps: coin tossing, marking, shuffling/sorting, and trimming. We prove the security in the presence of a semi-honest adversary that can corrupt $t < 1/2$ of the computing node P_i.

The ideal functionality f_R for the Resizer operator is described on Figure 1. To compute f_R securely, we implement the Resizer protocol π_R using a replicated secret-sharing scheme [1], where each computing party P_i has the following inputs as secret shares: dataset $[\mathcal{O}]$ of size N, attribute $[v]$ of size N, and success probability $[p]$, where the notation $[\cdot]$ represents a secret share. All computations are performed jointly by the parties using these secret-shared values, unless otherwise stated. Then, protocol π_R works as follows:

- **Coin Toss:** Each computing party P_i locally generates N random values $[rand]$ as secret shares of $rand$ as shown in [21] and computes a secret-shared bit-vector $[\ell] \in \{0,1\}^N$, where each entry is obtained via evaluating the comparison $[rand]_j < [p]$, representing the outcome of a single coin flip.
- **Mark:** Each computing party P_i computes a secret-shared bit-vector $[k] \in \{0,1\}^N$, by computing $[k]_j = [\ell]_j \vee [v]_j$.
- • **Shuffle:** Each computing party P_i permutes ($[\mathcal{O}]$, $[v]$, $[k]$) and obtains ($[\acute{\mathcal{O}}]$, $[\acute{v}]$, $[\acute{k}]$), **or**
  • **Sort:** Each computing party P_i sorts ($[\mathcal{O}]$, $[v]$, $[k]$) such that all rows j for which $k[j] = 0$ are at the bottom and obtains ($[\acute{\mathcal{O}}]$, $[\acute{v}]$, $[\acute{k}]$).
- **Trim:** Computing parties reveal (open) attribute $[\acute{k}]$ to obtain plaintext $\acute{k}$. They discard tuples in ($[\acute{\mathcal{O}}]$,$[\acute{v}]$), where $k[j] = 0$, resulting in the **output** dataset ($[\tilde{\mathcal{O}}]$,$[\tilde{v}]$) of size $S = T + \eta$.

Now, we prove that π_R securely computes the functionality f_R in the presence of a semi-honest adversary that corrupts less than half of the computing parties P_i.

Proof. Since the role of each computing party P_i in the secure computation is identical, we consider party P_1 (without loss of generality) to be corrupted by a semi-honest adversary $\mathcal{A}$. Consider a simulator $\mathcal{S}_R$, having as inputs, the input and output of P_1, i.e., $\mathcal{S}_R$'s inputs are the secret shares of dataset $[\mathcal{O}]$ of size N, attribute $[v]$ of size N, success probability $[p]$, and output dataset $[\tilde{\mathcal{O}}]$, and attribute $[\tilde{v}]$ of size $S = T + \eta$ each. The simulator then works as follows:

- $\mathcal{S}_R$ randomly selects a bit string $[\bar{\ell}]_{sim}$ of length N.
- $\mathcal{S}_R$ simulates the circuit evaluation of $\bar{k} = \bar{\ell} \vee v$ with party P_1. Then, P_1 receives output $[\bar{k}]$ and $\mathcal{S}_R$ receives output $[\bar{k}]_{sim}$.
- $\mathcal{S}_R$ simulates the oblivious shuffle (or sort) protocol with P_1 to shuffle $\bar{k}$, and simulates P_1's share to be $\left[\acute{\bar{k}}\right] = \acute{\bar{k}} \oplus \left[\acute{\bar{k}}\right]_{sim}$, where $\mathcal{S}_R$ selects $\acute{\bar{k}}$ such that it has S '1's, and selects $\left[\acute{\bar{k}}\right]_{sim}$ randomly. The shuffling of the dataset works as in a standard evaluation.
- $\mathcal{S}_R$ and P_1 together open the vector $\bar{k}$ and obliviously trim the database $\mathcal{O}$ to obtain $\tilde{\bar{\mathcal{O}}}$.

The simulated view of the party P_1 is $\left([\bar{k}], \left(\left[\acute{\bar{\mathcal{O}}}\right], [\bar{v}], \left[\acute{\bar{k}}\right]\right), \left(\left[\tilde{\bar{\mathcal{O}}}\right], [\tilde{\bar{v}}]\right)\right)$, and its real view is $\Big([k], ([\acute{\mathcal{O}}], [\acute{v}], [\acute{k}]), ([\tilde{\mathcal{O}}], [\tilde{v}])\Big)$. Since all the elements of the two views are secret shared, the shares itself are indistinguishable due to the security of the underlying MPC protocol. Moreover, the simulation forces the size of the output dataset to match that in the real world; thus, the size is identical in both views. Additionally, the choice between the shuffle and sort protocols does not affect the security of the Resizer, as the shuffled or sorted dataset is reshared, so the parties do not learn the new order of rows. Therefore, the Resizer protocol securely computes the ideal functionality f_R (Figure 1) in the presence of a semi-honest adversary.

7 Related Work

Chan et al. [10] and Qin et al. [31] both exploit the notion of differential obliviousness—a relaxation of full obliviousness that bounds the leakage of memory access patterns by a differentially private budget [10]. Chan et al. adopt an outsourced single-owner model, where one data owner delegates computation to an untrusted server and pads ORAM accesses with a DP-calibrated amount of dummy traffic, achieving substantial speed-ups while preserving formal privacy guarantees. In contrast, Qin et al. target a trusted hardware single-owner setting (Intel SGX); they inject random dummy accesses within the same address domain, thereby replacing the exact access patterns of physical relational operators with differentially oblivious counterparts calibrated at runtime. Both approaches differ fundamentally from the multi-owner, multi-party setting of our work.

Bater et al. [3] introduce Shrinkwrap, a framework for reducing the overhead of full oblivious padding in MPC-based secure query processing. For each query operator, a non-negative noise term η is drawn from a truncated Laplace distribution; the scale and truncation are set by a per-operator privacy budget that is obtained by splitting a global budget ϵ according to the operator's estimated selectivity. After the oblivious operator is evaluated, its true output cardinality T is computed, and the noisy size $T + \eta$ determines how many tuples are kept. The padded result is sorted so that genuine tuples appear first, and only the first $T + \eta$ tuples are forwarded to the next oblivious operator. Shrinkwrap is the closest prior approach that combines MPC with differential privacy to improve query-processing efficiency. However, Shrinkwrap provides *approximate* (ϵ, δ)-DP guarantees, whereas our construction achieves *pure* ϵ-DP (cf. § 2.2).

8 Implementation

We implement the Resizer on top of ORQ [4], a state-of-the-art, data-parallel engine for MPC-based SCA, rather than using a general-purpose MPC framework such as MP-SPDZ [22]. MP-SPDZ operates at a low level of abstraction and does not provide domain-specific optimizations for database analytics. In contrast, ORQ incorporates domain-specific optimizations and provides efficient primitives for sorting, shuffling, and comparisons. Further, ORQ identifies an upper bound on the size of the final aggregated `Join` result that is linear in the input size and independent of the specific data values. This bound reflects typical `Join` behavior observed in analytical workloads [4].

We extended ORQ's core library with two Resizer variants: **shuffle-based**, and **sort-based**. The two variants use the same biased Coin Toss and Mark steps, but differ in how they remove excess filler tuples. The complexities below are for implementing Resizer's steps in semi-honest, 3 Party Computation (3PC) with honest majority using secret-sharing protocols [1]. Table 1 summarizes Resizer's complexity.

Coin Toss & Mark. To sample uniform random value under MPC [21], each party locally generates a random Boolean share. The random value r is defined

as the XOR of all Boolean shares. Then, the shares are combined via XOR gates, which incur no communication cost, yielding a uniformly random secret without requiring interaction. The comparison and logic-OR operations are directly mapped to their MPC counterparts. This process is repeated N times; thus, the overall computational and communicational complexities remain $O(N)$, and the round complexity remains constant ($O(1)$) with respect to the number of rows N. This is achieved by ORQ vectorizing columnar data and executing operations in a single batch. Specifically, the batched logical-OR gates require only 1 communication round to evaluate across all N rows, and the comparison utilizes a prefix comparison circuit that completes in $O(\log x)$ rounds, where x is the bit length of our r (64 bits, 6 rounds).

Shuffle. ORQ realizes oblivious shuffling via shared permutations, which can be generated in a data-independent preprocessing phase and applied in the online phase without data-dependent control flow. This eliminates the need for expensive switching networks [18] and reduces both communication and computation to $O(M \cdot N)$ (with M denoting the tuple width in bits), while requiring only a constant number of rounds (3 in our 3PC setting).

Sort. ORQ optimizes Radixsort for a 1-bit sorting key. Thus, oblivious sorting reduces to a stable partitioning of tuples based on their sorting key bit values. ORQ computes oblivious prefix sums over the 1-bit sorting key to determine the destination indices of all '0's and '1's. These indices implicitly define a deterministic routing permutation, which is subsequently applied to data tuples using an oblivious routing primitive. While the prefix-sum computation requires $O(N)$ work, applying the permutation to N tuples of width M dominates the cost, resulting in an overall complexity of $O(N \cdot M)$, while requiring $O(\log_2 N)$ rounds.

Trim. In this step, the single-bit column k of length N is open. In ORQ, this requires $O(N)$ communication and computation and completes in a single round via batched share reconstruction. In the sort-based Resizer, after sorting, tuples with $k[j] = 0$ are placed at the end of the output table $\acute{\mathcal{O}}$ and are simply not passed as input to the subsequent operator. In the shuffle-based Resizer, *Trim* incurs the additional cost of copying the S tuples for which $k[j] = 1$ into a new table that serves as input to the next operator. This copying incurs a local computational cost of $O(M \cdot S)$, where S is the noisy IRS. This cost can be avoided by retaining the original table of size N and skipping tuples with $k[j] = 0$ in subsequent operators. However, this would require substantial modifications to the ORQ framework.

9 Evaluation

We conducted our evaluation using 3PC with an honest majority. The computing nodes are deployed on three servers with Intel(R) Xeon(R) Gold 5120 CPUs and

Table 1: Complexity of Resizer Steps in the ORQ framework [4]. N is the number of tuples or input size to the Resizer (worst-case padded IRS), M is the size of each tuple in bits, and S is the output size of the Resizer (Noisy IRS).

Operation	Computation Complexity	Communication Complexity
Comparator (Coin Toss)	$O(N)$	$O(N)$
Logical-OR (Mark)	$O(N)$	$O(N)$
Shuffle	$O(M \times N)$	$O(M \times N)$
1-bit Radixsort	$O(M \times N)$	$O(M \times N)$
Open 1-bit k (Trim)	$O(N)$	$O(N)$
Copy (Trim)	$O(M \times S)$	none

128 GB of RAM each. Our experiments are built on ORQ, inheriting its default settings and optimally tuned worker and communicator threads. We use the LAN (10Gb/s, 0.1ms RTT) and WAN emulation (10Gb/s, 20ms RTT) settings provided by ORQ.

In the following experiments, we answer three main questions:

1. How do the two Resizer variants perform as operators within ORQ in terms of execution time, and which variant is better for WAN?
2. What is the performance impact of placing the Resizer after relational operators (`Filter`, `Join`, or `GroupBy`)? What is the privacy-performance trade-off?
3. How does inserting a Resizer after each operator in an analytical query improve its runtime?

9.1 Resizer Performance

Experiment Setup. The dataset for this experiment is synthetically generated and organized in a three-column schema consisting of a 1-bit validity flag and two 64-bit integers. All columns are secret-shared via ORQ. The table size varies from 10^2 to 10^6 rows. This dataset serves as direct input to the Resizer. The Resizer's success probability p and its output do not affect the results and are therefore not further discussed in this experiment.

In this experiment, we compare the performance of the Resizer variants: the shuffle-based (Coin Toss, Mark, Shuffle & Trim) and the sort-based (Coin Toss, Mark, Sort & Trim). Figure 2 reports the runtime of both variants under a range of thread counts and network topologies (WAN vs. LAN). For both Resizer variants, all computations are done in the online phase, including the randomness generation for the *Coin Toss* and the generation of the permutation for the *Shuffle*.

In the WAN setting (left sub-Figure 2), which represents compute nodes administered by different entities, the results show that the shuffle-based Resizer consistently outperforms the sort-based variant. ORQ's 1-bit sorting requires more communication rounds than shuffling. In latency-dominated networks, this results in a clear benefit for the latter. In terms of ORQ's internal parallelism

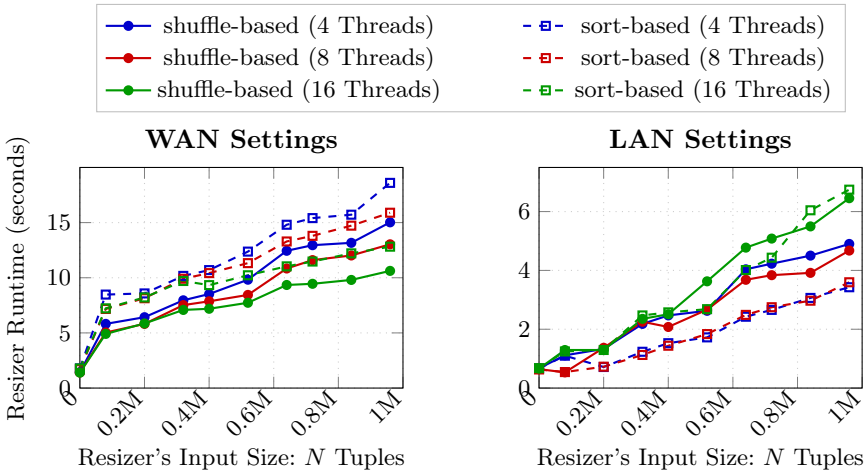

Fig. 2: Performance Comparison of shuffle-based Resizer (CoinToss-Mark-Shuffle-Trim) vs. sort-based Resizer (CoinToss-Mark-Sort-Trim) across varying network latencies (LAN: 0.1ms, WAN: 20ms) and thread count.

choice, both Resizer variants benefit from multiple threads, reaching their best runtimes with 16 threads.

In the LAN setting (right sub-Figure 2), the sort-based Resizer outperforms the shuffle-based, since the sort operation runs more efficiently in low-latency environments. In terms of the effect of internal parallelism on the Resizer variants, in LAN, the runtime does not improve much with additional threads beyond 4. For the sort-based variant, performance even degrades as the number of threads increases. This behavior is caused by the inherently sequential *Sort* and *Shuffle* steps, which become bottlenecks as more parallel resources are added.

9.2 Performance Impact of Resizer after Relational Operators.

Experiment Setup. We reused the synthetic dataset generated in the preceding experiment. The dataset serves as the input relation to the relational operator (`Join`, `Filter`, or `GroupBy`), whose output is then passed to the Resizer. The Resizer's success probability p and its output do not affect the results and are therefore not further discussed in this experiment. In this experiment, we evaluate the computational overhead introduced by the shuffle-based Resizer relative to oblivious relational operators. We performed the measurements on a WAN using 16 compute threads, and the communicator uses 4 connections.

Figure 3 shows that placing the Resizer directly after oblivious `Join` and `GroupBy` operators has only a minimal impact on the operator runtime. When placed after a `Filter` operator, the Resizer increases runtime, but when configured with a small p value, subsequent relational operators benefit from reduced intermediate result sizes.

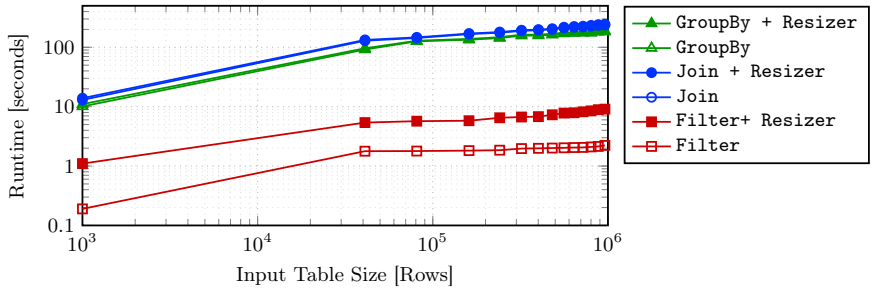

Fig. 3: Cost Comparison: Oblivious Operators vs. Operators + Shuffle-based Resizer. Both axes are on a logarithmic scale.

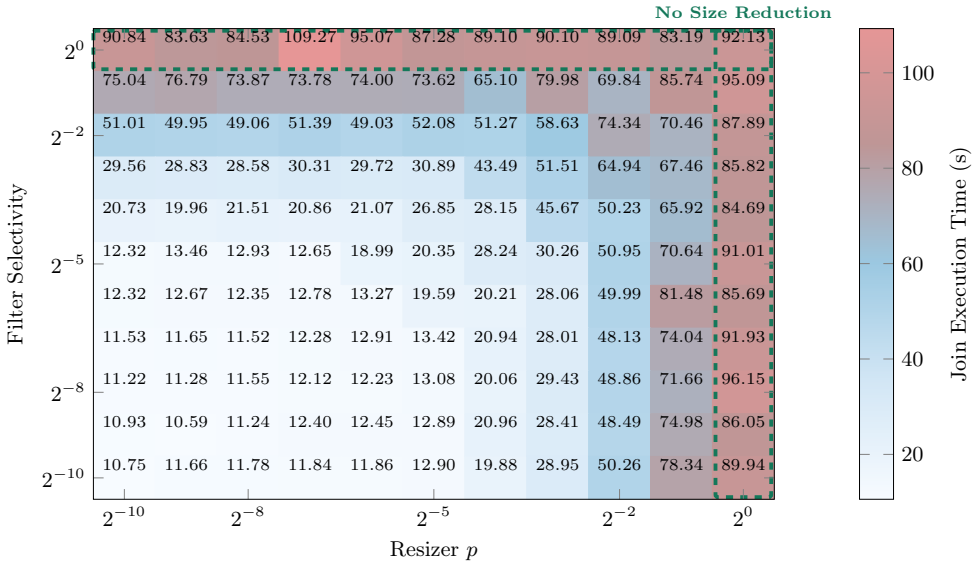

Fig. 4: Privacy-Performance Trade-off heatmap showing the oblivious Join execution time for `Qt` for varying p and selectivity for the `Filter + Resizer`. Reducing p can achieve up to $8-9\times$ reduction in Join execution time. However, reducing p to a value significantly less than 10^{-1} will not improve performance, even though it increases the privacy loss. For reference, the runtime of the `Filter + Resizer` operator is $1.95s$.

To evaluate the trade-off between privacy and query performance, we conducted a parameter sweep over a `Filter + Resizer`, varying the Filter's selectivity and the Resizer's success probability (p). To show the impact of reduced intermediate result size on a subsequent operator (oblivious `Join` without Re-

sizer), we ran the following query and measured the runtime of each operator:

```
Qt: SELECT * FROM table_l JOIN table_r
ON lt.key = rt.key WHERE lt.num < [limit]
```

We use a synthetic dataset: `table_l` has $100,000$ rows (N_L), `table_r` has 100 rows (N_R). To control the selectivity of the Filter, we populate `table_l` with sequential values and set `[limit]` using logarithmic steps ($N_L * 2^{-i}$ for $i = 0..10$). We set p also in logarithmic steps (2^{-i} for $i = 0..10$). We record the `Join` execution time of query `Qt` for all parameter combinations in a WAN setting with 16 threads (i.e., the same setup as in previous experiments using the shuffle-based Resizer).

Figure 4 visualizes the trade-off space as a heatmap. The fully oblivious `Join` baseline (the rightmost column, $p = 2^0$) incurs execution times of approximately 90 seconds. Because the ORQ `Join` complexity scales linearly with the input size ($O(N_L + N_R)$), $N_L = 100,000$ fully padded rows result in the maximum runtime regardless of the Filter's true selectivity. By reducing p, and given a Filter selectivity < 1, the Resizer shrinks the intermediate result size (N_L) passed to the `Join`. This results in almost an order-of-magnitude lower `Join` runtimes. Reducing p beyond $\sim 10^{-1}$, however, does not improve runtime anymore, while incurring a large privacy loss.

Based on this empirical evaluation, setting $p > 10^{-1}$ and at least $10\times$ the selectivity of the Filter appears to be a meaningful performance/security trade-off. The precise choice, however, remains a function of the total privacy budget and the overarching strategy of the parties running SCA.

9.3 End-to-End Query Performance with Resizer

Experimental Setup. To assess the impact of the shuffle-based Resizer on the overall performance of an end-to-end query, we conducted experiments on a WAN setup with 16 compute threads. We used the TPC-H benchmark [36] and set ORQ's built-in generator to produce a TPC-H compliant dataset with a scale factor of 1 (approximately 1GB of data) and up to 6 million rows per table. We run three highly complex, multi-way `Join` TPC-H queries—Q8, Q10, and Q21 (cf. § C. These queries were selected specifically for the exceptionally deep execution trees. In oblivious query execution, the deeper the pipeline, the greater the compounding penalty of processing padded dummy rows across successive operators. It is important to highlight that, in contrast to MP-SPDZ [22], in which `Joins` produce an output size equal to the Cartesian product of the inputs, ORQ identifies an upper bound on the size of the final aggregated `Join` result that is linear in the input size and independent of the specific data values. This bound reflects typical `Join` behavior observed in analytical workloads [4]. Nonetheless, even this smaller padding results in ballooning of intermediate result sizes in deep query trees. The Resizer is placed after every operator (except the last) in the query execution plan, with a success probability p of 0.1, favoring performance over privacy (i.e., on average, 10% of dummy rows are kept).

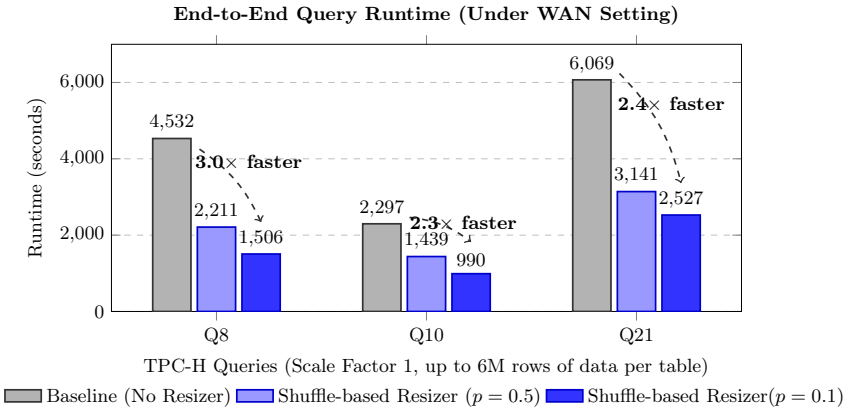

Fig. 5: End-to-End Query Execution Time for heavy TPC-H queries.

In the chosen TPC-H queries, filtering conditions typically retain only about 10% of the original rows. When filtered tables are joined, the number of successful matches similarly accounts for roughly 10% of the larger table size. The aggregation operators (`GroupBy`) condense the data drastically, often leaving around 1% of the padded block as valid output. Across the entire pipeline, the volume of the real, meaningful data averages roughly 10% of the intermediate results. Consequently, the post-Resizer intermediate results are approximately 20% of their worst-case table size. We selected this parameter to demonstrate the end-to-end performance benefits of removing the majority of fillers.

As shown in Figure 5, the Resizer achieves a speedup of up to 3× for Q8 and 2.4× for Q21, with Q10 showing a 2.3× improvement over the baseline ORQ system (oblivious operators). The additional execution cost of the shuffle-based resizer is quickly offset by the substantial reduction in intermediate result sizes. The benefits are particularly pronounced in queries with deep `Join` trees (e.g., Q8), where the cascading effect of reduced data volumes yields significant performance gains. These results demonstrate that, the Resizer provides a clear benefit by enabling more efficient downstream processing through early, aggressive pruning of dummy rows.

10 Conclusion

We introduce *Resizer*, a lightweight resizing/padding primitive that can be transparently integrated into relational query plans to reduce intermediate result sizes (IRS) from fully oblivious to differentially private (DP) noisy bounds. By leveraging a biased coin with success probability p, Resizer controls filler tuples in a manner analogous to operator selectivity while preserving formal DP guarantees. We present two practical variants—shuffle-based and sort-based—and establish their privacy via a Mangat-style randomized response construction, along with

security guarantees in the semi-honest model. Our evaluation shows that these guarantees come at negligible cost. Across both LAN and WAN settings, Resizer consistently reduces intermediate data volume and improves end-to-end performance. In particular, complex TPC-H queries in the ORQ framework achieve substantial speedups, reaching up to 3× in our specific WAN experiments compared to the baselines.

Acknowledgments. This project received funding from the European Research Council (ERC) under the European Union's research and innovation programs Horizon Europe (PRIVTOOLS/101124778) and Horizon 2020 (PSOTI/850990). Additionally, it was co-funded by the Deutsche Forschungsgemeinschaft (DFG) within within SFB 1119 CROSSING/236615297, as well as by the German Federal Ministry of Research, Technology and Space (BMFTR) and the Hessian Ministry of Science and Research, Arts and Culture (HMWK) as part of the National Research Center for Applied Cybersecurity ATHENE.

References

1. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2016)
2. Bater, J., Elliott, G., Eggen, C., Goel, S., Kho, A., Rogers, J.: SMCQL: secure querying for federated databases. Proc. VLDB Endow. (2017)
3. Bater, J., He, X., Ehrich, W., Machanavajjhala, A., Rogers, J.: Shrinkwrap: efficient SQL query processing in differentially private data federations. Proc. VLDB Endow. (2018)
4. Baum, E., Buxbaum, S., Mathai, N., Faisal, M., Kalavri, V., Varia, M., Liagouris, J.: ORQ: Complex analytics on private data with strong security guarantees. In: Proceedings of the ACM Symposium on Operating Systems Principles. ACM (2025)
5. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Annual International Cryptology Conference. Springer (1991)
6. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: Proceedings of the Annual ACM Symposium on Theory of Computing (1990)
7. Ben-Efraim, A., Lindell, Y., Omri, E.: Optimizing semi-honest secure multiparty computation for the internet. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2016)
8. Ben-Efraim, A., Nielsen, M., Omri, E.: Turbospeedz: Double your online spdz! improving spdz using function dependent preprocessing. In: International Conference on Applied Cryptography and Network Security. Springer (2019)
9. Bogatov, D., Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: ϵpsolute: Efficiently querying databases while providing differential privacy. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. ACM (2021)
10. Chan, T.H.H., Chung, K.M., Maggs, B., Shi, E.: Foundations of differentially oblivious algorithms. J. ACM (2022)
11. Chaudhari, H., Choudhury, A., Patra, A., Suresh, A.: Astra: High throughput 3pc over rings with application to secure prediction. In: Proceedings of the ACM SIGSAC Conference on Cloud Computing Security Workshop (2019)

12. Chu, S., Zhuo, D., Shi, E., Chan, T.H.H.: Differentially oblivious database joins: Overcoming the worst-case curse of fully oblivious algorithms (2021)
13. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Annual International Cryptology Conference. Springer (2012)
14. Demmler, D., Schneider, T., Zohner, M.: ABY-A framework for efficient mixed-protocol secure two-party computation. In: Network and Distributed System Security Symposium (2015)
15. Doerner, J., Shelat, A.: Scaling oram for secure computation. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. ACM (2017)
16. Dwork, C., McSherry, F., Nissim, K., Smith, A.: Calibrating noise to sensitivity in private data analysis. In: Proceedings of the Conference on Theory of Cryptography. Springer-Verlag (2006)
17. Gu, L., Zeitouni, S., Binnig, C., István, Z.: Reflex: Speeding up smpc query execution through efficient and flexible intermediate result size trimming (2025)
18. Holland, W., Ohrimenko, O., Wirth, A.: Efficient oblivious permutation via the waksman network. In: Proceedings of the ACM on Asia Conference on Computer and Communications Security. ACM (2022)
19. Johnson, N., Near, J.P., Song, D.: Towards practical differential privacy for SQL queries. Proc. VLDB Endow. (2018)
20. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. ACM (2016)
21. Keller, H., Möllering, H., Schneider, T., Tkachenko, O., Zhao, L.: Secure noise sampling for DP in MPC with finite precision. Cryptology ePrint Archive, Paper 2023/1594 (2023)
22. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2020)
23. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free xor gates and applications. In: International Colloquium on Automata, Languages, and Programming. Springer (2008)
24. Koti, N., Pancholi, M., Patra, A., Suresh, A.: SWIFT: Super-fast and robust Privacy-Preserving machine learning. In: USENIX Security Symposium. USENIX Association (2021)
25. Liagouris, J., Kalavri, V., Faisal, M., Varia, M.: SECRECY: Secure collaborative analytics in untrusted clouds. In: USENIX Symposium on Networked Systems Design and Implementation. USENIX Association (2023)
26. Lutsch, A., El-Hindi, M., Heinrich, M., Ritter, D., István, Z., Binnig, C.: Benchmarking analytical query processing in intel sgxv2. In: Proceedings of the International Conference on Extending Database Technology. OpenProceedings.org (2025)
27. Mangat, N.S.: An improved randomized response strategy. Journal of the Royal Statistical Society: Series B (Methodological) (1994)
28. Micali, S., Goldreich, O., Wigderson, A.: How to play any mental game. In: Proceedings of the ACM Symp. on Theory of Computing. ACM (1987)
29. Mohassel, P., Rindal, P.: ABY3: a mixed protocol framework for machine learning. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. ACM (2018)

30. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2. 0: Improved mixed-protocol secure two-party computation. In: USENIX Security Symposium. USENIX Association (2021)
31. Qin, L., Jayaram, R., Shi, E., Song, Z., Zhuo, D., Chu, S.: Adore: Differentially oblivious relational database operators. Proc. VLDB Endow. (2022)
32. Rosulek, M., Roy, L.: Three halves make a whole? beating the half-gates lower bound for garbled circuits. In: Annual International Cryptology Conference. Springer (2021)
33. Sohn, D., Jiang, K., Hammer, N., Rogers, J.: Alchemy: A query optimization framework for oblivious sql. Proc. VLDB Endow. (2025)
34. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. CCS '13, ACM (2013)
35. Takagi, S., Kato, F., Cao, Y., Yoshikawa, M.: Asymmetric differential privacy. In: IEEE International Conference on Big Data. IEEE (2022)
36. Transaction Processing Performance Council (TPC): TPC Benchmark H (TPC-H) Standard Specification, Revision 3.0.1 (2022), https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf
37. Vadhan, S.P.: The complexity of differential privacy. In: Tutorials on the Foundations of Cryptography. Springer International Publishing (2017)
38. Yao, A.C.C.: How to generate and exchange secrets. In: Annual Symposium on Foundations of Computer Science. IEEE (1986)
39. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer (2015)

A Secure Database Systems with MPC

In the relational model, datasets are organized according to a schema. A *schema* defines the structure of the data, specifying a set of *relations* (tables). Each relation consists of *attributes* (columns), which describe the type of information stored, and *tuples* (rows), which represent individual records. To uniquely identify tuples, relations often include keys: a primary key is an attribute (or set of attributes) that uniquely identifies each row, while foreign keys establish relationships between tables by referencing primary keys in other relations. Queries over this data are expressed using relational operators such as selection, projection, joins, and aggregation.

MPC-based Secure Collaborative Analytics (SCA) enables multiple mutually distrustful parties to jointly evaluate queries over their private inputs while revealing only the prescribed outputs. The main requirement is oblivious execution, meaning that the protocol's observable behavior—computation, memory access patterns, communication volume, and timing—must be independent of private data. Accordingly, relational operators are implemented using oblivious primitives (e.g., comparisons, arithmetic operations, and sorting) whose control flow does not depend on input values, and a complete query is assembled by composing these primitives.

To store operator's input/output relations, Oblivious RAM (ORAM) is used, which ensures that memory accesses to tuples are indistinguishable [34,15]. However, achieving full obliviousness requires accounting for multiple leakage channels. While ORAM effectively hides access patterns, it does not hide communication volume or execution time, which a passive adversary can exploit to infer sensitive information. In particular, the execution time of an oblivious primitive/operator and the volume of communication reveal the Intermediate Result Size (IRS). Note that the IRS of an operator is determined by both the underlying data and the operator's (public) predicate (§ 3), so any variation in runtime or message size can directly leak information about the underlying private data, e.g., distribution of values. Recent work has shown that leakage of execution time or result sizes enables reconstruction attacks [20].

To eliminate leakage via IRS, systems pad intermediate results to a fully oblivious size, typically corresponding to the worst-case output cardinality. This yields a uniform, scan-based execution model that precludes index- or hash-based operator implementations.

Operators with predicates evaluate the condition on each input tuple and store the result in an extra column, a *validity mask* v. For each tuple, this secret bit is set to 1 if the condition is true and 0 otherwise. The validity mask is naturally padded to the worst-case of N, similar to the output relation size. For joins, the worst-case cardinality N equals the size of the Cartesian product of the two input tables, each of size $|R|$ and $|S|$, resulting in $|R \times S|$ rows. Therefore, each operator propagates the worst-case padded intermediate result, including the validity mask, bringing all redundant, non-matching tuples into downstream operators. Full ORAM plus padding incurs prohibitive overhead and scales only to small datasets (hundreds of tuples) [2]. However, when all tuples are accessed uniformly, the memory access pattern becomes data-independent, allowing ORAM to be omitted without additional leakage. This relaxation dramatically improves performance, enabling MPC-based query execution on hundreds of thousands of rows (e.g., 295K rows in 6.5h) [25].

B Resizer Privacy for Operator *op*.

Analogous to the analysis of privacy in § 5, in this section we analyze the privacy for any operator *op* with a sensitivity s. For q-neighbouring (Definition 4) datasets D and D', the true result sizes T and T', respectively, have the following

relation: $T = T' - s$. Then, using the Resizer mechanism, we get

$$\frac{Pr[M(D) = S]}{Pr[M(D') = S]} = \frac{Pr[M(D) = T + \eta]}{Pr[M(D') = T' + \eta']} == \frac{Pr[Z = \eta]}{Pr[Z = \eta - s]} \tag{7}$$

$$= \frac{\binom{N}{\eta} \beta^{\eta}(1-\beta)^{(N-\eta)}}{\binom{N}{\eta - s} \beta^{\eta - s}(1-\beta)^{(N-(\eta - s))}} \tag{8}$$

$$= \left[\prod_{i=0}^{s-1} \frac{N - \eta + s - i}{\eta - i}\right] \cdot \frac{\beta^s}{(1-\beta)^s}. \tag{9}$$

Now, solving for $R(\eta) = \left[\prod_{i=0}^{s-1} \frac{N - \eta + s - i}{\eta - i}\right] \cdot \frac{\beta^s}{(1-\beta)^s}$ for $\eta = s$, $R_{max} = \left[\prod_{i=0}^{s-1} \frac{N - i}{s - i}\right] \cdot \frac{\beta^s}{(1-\beta)^s}$, and for $\eta = N$, $R_{min} = \left[\prod_{i=0}^{s-1} \frac{s - i}{N - i}\right] \cdot \frac{(1-\beta)^s}{\beta^s}$. Then, the mechanism satisfies differential privacy when

$$\epsilon \geq \ln\left(\max\left(R_{max}, 1/R_{min}\right)\right) = \ln\left(\left[\prod_{i=0}^{s-1} \frac{N - i}{s - i}\right] \cdot \max\left(\frac{\beta^s}{(1-\beta)^s}, \frac{(1-\beta)^s}{\beta^s}\right)\right). \tag{10}$$

Similar to the case of sensitivity 1, the best privacy is achieved when $\beta = 1/2$. And, the selection of parameters for increased privacy or performance follows the description in § 5.1.

Note: For the `Join` operator, the sensitivity is $s = m$, where m is the size of the second table. And, for the `GroupBy` operator, the sensitivity is $s = 2$.

C TPC-H Queries

Listing 1.1: TPC-H Query Q8 – Market-share per year

```
SELECT o_year, SUM(CASE
                  WHEN nation = '[NATION]' THEN volume ELSE 0
                  END) / SUM(volume) AS mkt_share
FROM (
    SELECT EXTRACT(YEAR FROM o_orderdate) AS o_year, l_extendedprice * (1 - l_discount) AS
         volume, n2.n_name AS nation
    FROM part, supplier, lineitem, orders, customer, nation n1, nation n2, region
    WHERE p_partkey = l_partkey AND s_suppkey = l_suppkey AND l_orderkey = o_orderkey AND
         o_custkey = c_custkey AND c_nationkey = n1.n_nationkey AND n1.n_regionkey =
         r_regionkey AND r_name = '[REGION]' AND s_nationkey = n2.n_nationkey AND o_orderdate
          BETWEEN DATE '1995-01-01' AND DATE '1996-12-31' AND p_type = '[TYPE]'
) AS all_nations
GROUP BY o_year ORDER BY o_year;
```

Listing 1.2: TPC-H Query Q10 – Revenue per customer

```
SELECT c_custkey, c_name, SUM(l_extendedprice * (1 - l_discount)) AS revenue, c_acctbal,
     n_name, c_address, c_phone, c_comment
```

```
FROM customer, orders, lineitem, nation
WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate >= DATE '[DATE]' AND
     o_orderdate < DATE '[DATE]' + INTERVAL '3' MONTH AND l_returnflag = 'R' AND c_nationkey
      = n_nationkey
GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment
ORDER BY revenue DESC;
```

Listing 1.3: TPC-H Query Q21 – Suppliers with the most waiting orders

```
SELECT s_name, COUNT(*) AS numwait
FROM supplier, lineitem AS l1, orders, nation
WHERE s_suppkey = l1.l_suppkey AND o_orderkey = l1.l_orderkey AND o_orderstatus = 'F' AND l1.
     l_receiptdate > l1.l_commitdate
    AND EXISTS (
        SELECT * FROM lineitem AS l2
        WHERE l2.l_orderkey = l1.l_orderkey AND l2.l_suppkey <> l1.l_suppkey
      )
   AND NOT EXISTS (
        SELECT * FROM lineitem AS l3
        WHERE l3.l_orderkey = l1.l_orderkey AND l3.l_suppkey <> l1.l_suppkey AND l3.
             l_receiptdate > l3.l_commitdate
      )
   AND s_nationkey = n_nationkey AND n_name = '[NATION]'
GROUP BY s_name ORDER BY numwait DESC, s_name;
```