Reflex: Faster Secure Collaborative Analytics via Controlled Intermediate Result Size Disclosure

Long Gu
Systems Group, TU Darmstadt, Germany

Shaza Zeitouni
Systems Group, TU Darmstadt, Germany

Carsten Binnig
Systems Group, TU Darmstadt, Germany

Zsolt István
Systems Group, TU Darmstadt, Germany

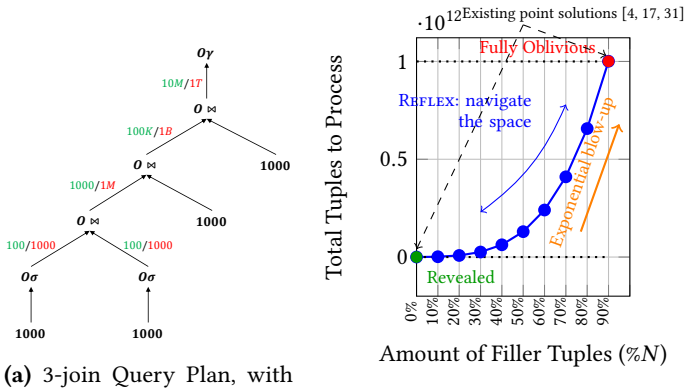
Abstract

Secure Multi-Party Computation (MPC) enables collaborative analytics without exposing private data. However, OLAP queries under MPC remain prohibitively slow due to oblivious execution and padding of intermediate results with filler tuples. We present Reflex, the first framework that enables *configurable trimming of intermediate results across different query operators*—joins, selections, and aggregations—within full query plans. At its core is the *Resizer* operator, which can be inserted between any oblivious operators to selectively remove filler tuples under MPC using user-defined probabilistic strategies. To make privacy trade-offs interpretable, we introduce a new metric that quantifies the number of observations an attacker would need to infer the true intermediate result sizes. Reflex thus makes the performance–privacy space of secure analytics *navigable*, allowing users to balance efficiency and protection. Experiments show substantial runtime reductions while maintaining quantifiable privacy guarantees.

1 Introduction

The analysis of internal data is a critical component of strategic decision-making in large organizations. Since most of these organizations operate globally, transferring and processing data across countries and jurisdictions without strong cryptographic protection introduces risks and may even be restricted by regulations such as the GDPR [19]. As a result, there is a growing effort [4, 31, 42, 44] to combine analytical data processing with Secure Multi-Party Computation (MPC) in order to provide stronger security and privacy guarantees. MPC allows several parties with private data, e.g., company branches, to jointly compute a function, e.g., a database query, without any party learning more in the process than the final result.

However, performing computations under MPC is several orders of magnitude more expensive than in plaintext [4, 23, 31, 37]. In addition to the per-operation computation overhead, there is a compounding factor for OLAP-like workloads: algorithms in MPC must execute *obliviously*, i.e., without revealing any information about the underlying data. For queries, this means that intermediate results passed between operators of a query plan must be padded with filler tuples to the maximum possible size, preventing attackers from inferring information based on the selectivity of the operators. For example, selection operators must return an output of the same size as the input, and joins must produce an output with the size of the Cartesian product of both tables. Overall, for query plans with multiple operators, this can lead to result size exploitation as shown in Figure 1a.



(a) 3-join Query Plan, with (red) and without (green) oblivious output sizes

(b) 3-join's total tuples processed under varying strategies.

Figure 1. Motivating 3-join example: all operator selectivities are fixed at 10%, and the total number of tuples to process (sum of all intermediate result sizes) is a function of the amount of oblivious filler tuples included in each intermediate result. The trend is exponential when going from *no fillers* (green) to *fully oblivious intermediate sizes* (+90%N filler tuples at each operator, red)– this explains the severe performance penalty under fully-oblivious MPC.

In this paper, we present Reflex, a novel approach that accelerates private query processing for OLAP workloads.The key idea behind Reflex is to make the *performance–privacy trade-off* in query execution both visible and controllable. It is the first method that allows even non-experts to *quantitatively* explore how much performance can be gained for a given level of privacy protection. At the heart of Reflex is a new query operator called the *Resizer*. This operator can be placed between any pair of oblivious operators in a query plan. It reduces the size of intermediate results by *randomly trimming filler tuples*—dummy records added for privacy. The trimming amount is determined through *secure*

sampling under MPC, guided by user-defined probability distributions that capture different performance–privacy preferences. To measure the amount of privacy each distribution provides, we introduce a new metric that quantifies how many repeated observations of the trimmed intermediate result sizes an attacker would need to accurately infer the true intermediate result size.

The concept of trimming intermediate results has been explored in several recent works. However, existing approaches have primarily focused on the *join* operator [5, 9] and do not integrate all mechanisms into a single unified framework to make privacy quantifiable at the query plan level. To the best of our knowledge, there is currently no *general mechanism* that applies uniformly across different query operators (e.g., join, selection, aggregation), integrates seamlessly into query plans without modifying existing oblivious operators, and enables fine-grained control over the performance–privacy trade-off at the operator level. Moreover, prior works typically employ fixed trimming strategies tailored to specific privacy assumptions, such as differentially private resizing [5]. We argue that a practical analytics framework should instead provide both *flexibility* and a *unified means* to quantify the privacy guarantees of user-defined trimming strategies for intermediate result sizes.

To achieve the goals of Reflex, we address two key challenges in this paper. First, it requires a *generic, configurable, and efficient* resizing mechanism that can be integrated into diverse query plans. The Resizer must yield tangible performance gains by reducing upstream computation, without introducing overheads that offset these benefits. Achieving both configurability and efficiency across heterogeneous operators demands careful algorithmic and system-level co-design. Second, Reflex must address a *usability challenge*: how to provide non-security experts with an intuitive means to quantify and navigate the performance–privacy trade-off introduced by trimming filler tuples. To this end, we develop a practical metric that abstracts away cryptographic complexity while capturing how resizing choices affect both system efficiency and potential leakage of intermediate result sizes.

Addressing these challenges within a unified framework enables a principled exploration of the performance–privacy space, moving beyond static, one-size-fits-all oblivious execution. As illustrated later in Figure 11, different placements and configurations of the Resizer yield predictable trade-offs between runtime and privacy. Our experiments show that trimming can reduce query runtimes by up to an order of magnitude, while an attacker would still require multiple rounds of observation to infer the true intermediate sizes. Ultimately, this line of work can pave the way for *MPC-aware query optimizers* that jointly explore the space of oblivious query plans and resizing strategies to find configurations that maximize performance while meeting certain user-defined security guarantees. Building such an optimizer, however, is beyond the scope of this paper. To summarize, our contributions are as follows:

- **A first framework for privacy–performance trade-offs**. We present Reflex, a framework that provides fine-grained, user-controlled trade-offs between performance and privacy for Secure collaborative analytics (SCA). The central primitive is the Resizer, a lightweight operator that can be inserted after any oblivious operator to selectively remove filler tuples according to user-specified trimming strategies while preserving obliviousness. Resizer's MPC-friendly design increases parallelism and reduces communication rounds, enabling efficient and flexible reduction of intermediate result sizes. We implement a prototype and release the code as open-source; see Section 6.
- **A statistically grounded metric.** To enable principled comparison between different user-defined trimming approaches, we propose a metric grounded in statistical methods that quantifies how many observations of trimmed intermediate result sizes an attacker would need to recover the true intermediate result size with high probability. The metric supports direct, interpretable comparisons between different trimming strategies and can be used to place points in the privacy–performance design space.
- **A comprehensive evaluation**. We thoroughly evaluate Reflex through micro-benchmarks, reimplementation & comparison with related methods [5, 17], as well as full query experiments using analytical queries from prior work. Our study characterizes runtime across multiple points in the trade-off space, and demonstrates the practical gains and limits of different trimming strategies under both performance and privacy metrics.

2 Background and Related Work

2.1 Secure Collaborative Analytics

Secure collaborative analytics (SCA) [31] is a set of distributed data statistics protocols designed to protect private datasets provided by *Data Owners*. Let $\mathcal{DO} = \{DO_1, \ldots, DO_\ell\}$ be a set of ℓ *Data Owners*, where each DO_i holds a private database D_i. Secure Collaborative Analytics protocols enable joint data analysis on the union of all private datasets $\mathcal{D} = \{D_1, \ldots, D_\ell\}$ while safeguarding against privacy violations. Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ denote a set of m computing nodes or parties that are responsible for executing secure collaborative analytics protocols on $\mathcal{D}$. The *Data Analyst* sends the query Q to be executed over the union of all private datasets $\mathcal{D}$ to $\mathcal{P}$ and collects the final result R. Overall, SCA is a growing field with numerous real-world applications, including market analysis, autonomous driving, agriculture, and the Internet of Things. These applications benefit from

efficient query execution among multiple data owners, which also preserves data privacy [41].

2.2 Related Work using MPC

To the best of our knowledge, there is no related work that presents a flexible framework for balancing the trade-off between revealing information about intermediate result sizes in query execution in a semi-honest setting. The related work in SCA using MPC can be grouped into one of three groups: first, those that aim to optimize the underlying oblivious operators given metadata about the query, second, those that offer point solutions in terms of relaxing security guarantees, and third, those that aim to protect against a much stronger, malicious attacker.

Related work, such as SMCQL [4], Secrecy [31], Senate [44], and Alchemy [40], aims to increase performance by leveraging relations' metadata and redesigning actual computation (hence also query execution) under MPC or applying algorithmic changes to the MPC protocols without loosening security/privacy guarantees. One of the most recent works in this direction is Alchemy [40]. Alchemy optimizes queries through rewrite rules, cardinality bounds, bushy plan generation, and a fine-grained cost model, minimizing circuit complexity while maintaining security guarantees. Such optimizations are orthogonal to our work and can be adopted in Reflex to further improve performance.

Other related work, such as Secretflow [17], Shrinkwrap [5], and SAQE [6], improves performance by relaxing the privacy guarantees, such as information about the intermediate result size of operators [5, 17], or the tuples that form the intermediate result [17]. Some works even allow for relaxing correctness of query results [5, 6]. Closest to the approach of Reflex is Shrinkwrap [5], which introduces a point solution that reduces fully-oblivious intermediate result sizes to differentially private sizes. This is achieved with a "sort&cut" approach, which couples trimming with a potential reduction in query accuracy (no guarantee of correctness). In the context of analytics within an enterprise, this is not desirable – in Reflex, queries always produce the same results, regardless how many filler tuples have been removed from the intermediate result.

SAQE [6] employs a private sampling algorithm that trades correctness for performance, providing a scalable and fast approximate query processing platform.

Finally, there is also related work that focuses not on performance improvements but on protecting against a stronger attacker in the malicious security model, e.g., Senate [44] and Scape [22]. Since protocols that are secure against a malicious attacker are yet again orders of magnitude slower than those for semi-honest systems [13, 18], and since these systems introduce restrictive assumptions about how data can be processed, they are not a perfect match for the enterprise-level analytics we are focusing on.

2.3 Related Work using TEEs

In case the security model permits a trusted third party, Trusted Execution Environments (TEEs), such as Intel SGX [11, 34] or AMD SEV [26], can be used to perform fast analytics on shared data. Sensitive data is encrypted and only decrypted and processed within the TEE. There is rich related work, such as EnclaveDB [36], ObliDB [16], and benchmarking papers [33], showing that TEEs can be used for data analytics tasks with minimal performance overhead compared to plaintext execution. Compared to MPC, TEE-based methods are orders of magnitude faster. However, their adoption is restricted to the use of specialized hardware and vulnerabilities such as side-channel attacks [8, 12, 30, 39].

2.4 Related Work using FHE

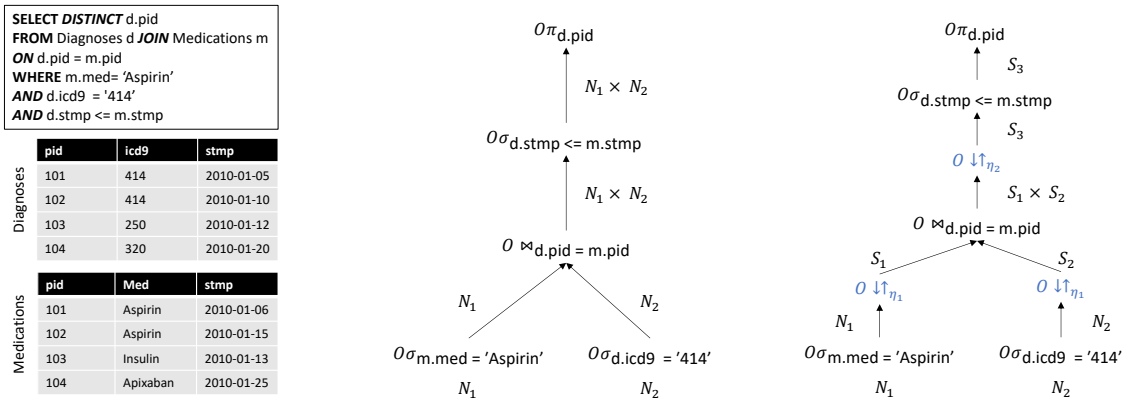
There are also related works based on Fully Homomorphic Encryption (FHE), such as HE3DB [7]. It allows multiple data owners to encrypt their private input and conduct collaborative analytics in the cloud server without exposing intermediate results. HE3DB [7] utilizes the latest advancements in FHE [21] and allows for server-side elastic analytical processing of requested FHE ciphertexts, including private decision tree assessment, unlike previous encrypted DBMS that only provide aggregated information retrieval. While FHE provides stronger security guarantees than MPC, sparking growing interest, its significant performance overhead currently renders it impractical for complex operations, such as secure sorting or joins, which are at the core of SCA.

3 Reflex: Assumptions and Threat Model

In SCA, the goal is to protect the privacy of input datasets and computation. Ideally, no party should learn more about the data than what can be inferred from the final query result.

In the deployment model as described in Section 2.1, each data owner, DO_i, distributes secret shares of their dataset, D_i, to the computing nodes. A query Q 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 under MPC, all parties, i.e., data owners, computing nodes, and the data analyst, are assumed to know the database schema and the queries to be executed. During the execution of π on input data shares of $\mathcal{D}$, each computing node P_i has a view that includes the following:

- P_i's secret share of $\mathcal{D}$ and any locally generated random values required by the protocol,
- messages received during protocol execution,
- P_i's secret shares of intermediate results,
- *trimmed* sizes of intermediate results. These are some values between true intermediate result size and fully-oblivious result size, and
- P_i's secret shares of the final response R.



(a) Example Query **(b)** Fully Oblivious Query Plan **(c)** Oblivious Query Plan with Resizer (⇵)

Figure 2. A Query Example and its Plans. O indicates oblivious operators, N_i is input/output sizes, and S_i refers to re-sized output after ⇵ operator with ($S_i < N_i$).

We assume a *semi-honest* trust model, where computing nodes follow the protocol honestly but may attempt to infer information from their local view about the input datasets, intermediate results, or the sizes of intermediate results of query operators. This trust model is suitable for protecting against accidental data leaks, such as logs exposed to system administrators or lost disks. and it aligns well with the context of a large corporation (similar to the use cases presented in [17, 41]), where data owners and computing nodes belong to branches or departments within the same large corporation. The goal of using secure MPC in this context is to protect the privacy of computation and data in motion, to contribute to data protection by design [10].

On top of the semi-honest trust model, we adopt the following adversary model, where: *(i)* at most one computing node may be corrupted, and *(ii)* parties do not collude. Therefore, it is reasonable to assume an attacker can *only* observe, without invasive interference, the trimmed intermediate result sizes of all operators due to the insertion of Resizer operators across queries executed in the system and use these observations of trimmed intermediate result sizes to learn the true output size of an operator.

As a way to quantify the information leakage as a result of trimming filler tuples from intermediate results, we introduce the *Rounds to Recover* metric (detailed in Section 4.4) that determines the number of equivalent repetitions of an operator required to infer its true result size with high probability. An equivalent repetition refers to any execution, possibly across different queries, where the operator processes the same input and produces the same intermediate result. This metric provides non-security experts with an intuitive way to compare the trade-off between performance and privacy of different strategies that Reflex allows them to define. Note that in this work, we do not study the effect of revealing trimmed intermediate result sizes on the privacy of datasets. Therefore, in the remainder of the paper, by 'privacy' we mean the privacy of intermediate result sizes.

4 Reflex: Design and Implementation

In the following, we provide an overview of Reflex before delving into the details of its core mechanisms and the metric used to quantify the security gains.

4.1 Overview of Reflex

Reflex is a framework that provides fine-grained, user-controlled trade-offs between performance and privacy for SCA. This is achieved with the help of Resizer operators that can be inserted after any oblivious operator to remove filler tuples according to a user-specified trimming strategy. Resizer has been implemented in a way that can take advantage of communication batching optimizations in MPC, with a runtime comparable to oblivious filter, join, and group by operators. See Section 4.2 for more details.

One of the major novelties in Reflex is the flexibility of deciding how many filler tuples are kept, without having to change the underlying Resizer implementation. Figure 2 illustrates a simple query where Resizer operators are inserted to reduce the output size of the filters following a scan and the output size of a join. Flexibility in configuration and placement enables Reflex to achieve significant performance improvements.

This flexibility of placement is achieved by plugging in a user-defined strategy into each Resizerinstance. The strategy is, at its core, a pre-defined probability distribution for determining how many filler tuples to keep in the intermediate result passed to the next operator. Note that the distribution can be re-configured as often as needed to fulfill performance or privacy requirements. In Section 4.3, we discuss the properties a distribution should have to ensure some protection of

the intermediate result size, and present concrete examples of distributions and how they are sampled.

Without a straightforward way to compare user-defined distributions, it is not really possible to argue for specific ones. To this end, in Section 4.4 we present the metric we propose to compare distributions in terms of how much information they leak.

Our Implementation: For our implementation, we utilize MP-SPDZ [27], a powerful framework for secure MPC, which compiles applications written in a Python-like code into MPC protocols. In addition to offering state-of-the-art performance, MP-SPDZ enables the selection of different threat models and underlying cryptographic primitives without requiring changes to the code.

To achieve efficient and secure computation, we choose Replicated Secret Sharing (RSS) [2] as the underlying secret sharing scheme. RSS is highly efficient because it requires only a single round of communication for basic arithmetic operations, offering low latency compared to other protocols. Its integration within MP-SPDZ allows for seamless deployment and execution. MP-SPDZ employs a mixed domain execution model. This allows the system to seamlessly convert secret shares between the arithmetic and binary (Boolean) domains using edaBits [14]. The internal compiler/optimizer automatically determines which domain is most efficient for executing a given function.

In addition to the Resizer operator, our prototype provides independent implementations of the following operators:

- Scan: This operator reads private data into a secret shared matrix using for loop in MP-SPDZ [27]. It assumes a known, fixed size for the dataset, allowing secure access and processing of each element. Note that a scan is always coupled with one of the other operators.
- Filter: It applies a condition (multiple equality tests) to the entire table in a for loop, creating a new secret shared column marking matches.
- Join: Implemented as a nested-loop join (NLJ), this operator uses two nested loops to compare every row from the first table (size n) with every row from the second table (size m). The column on which to evaluate the join predicate in each table is configurable. Joins produces an output of size $n * m$ and an extra secret column that marks true join result tuples.
- Group By: Similarly to the related work [4, 31], we use a sorting-based method. We first order rows by the group key. Then, iterates row by row, comparing adjacent keys over secrets to identify group boundaries, marking them accordingly for subsequent aggregation operations.

Queries are hand-assembled trees of SQL operators and instances of Resizer. Automated translation of plaintext SQL queries into MPC protocols remains future work.

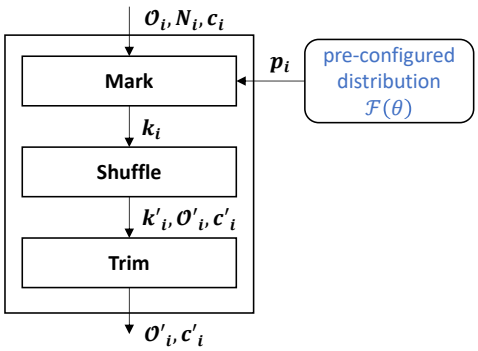


Figure 3. Resizer operator. Inputs: oblivious output O_i, oblivious output size N_i, true output column c_i of operator O_i. Outputs: shuffled output O'_i and c'_i indicating true tuples.

4.2 Resizer: A Helper Operator

A Resizer operator can be inserted after any oblivious operator to reduce its output size by discarding filler tuples partially, without requiring query rewrites. It is built as a sequence of three processing steps: Mark, Shuffle and Trim (see Figure 3), which we detail in their own subsections. The Resizer runs under MPC and reveals no information about the contents of the output of the operator it runs after, apart from the final *trimmed* output size $S_i = T_i + \eta_i$ where $T_i \leq S_i \leq N_i$ and η_i is the amount of filler tuples retained by the Resizer.

The amount of filler tuples to keep at each instance of Resizer is decided at run-time (under MPC) using a pre-configured user-defined function (i.e., a distribution) that leverages both publicly-available information, e.g., input sizes and the operator trees, and secret information.

4.2.1 How we modify the output of oblivious operators?

An oblivious operator O_i in a query plan produces an oblivious output $O_i = O_i(\mathcal{D})$ of size N_i (i indexes the operator in the query tree, e.g., Figure 2c). By construction, this output comprises T_i tuples that are genuine output (called interchangeably "true matches") of the operator, while the remaining $N_i - T_i$ tuples are indistinguishable fillers that are retained as part of the output to satisfy obliviousness; we refer to these as *filler tuples*.

The Resizer (see Figure 3) takes as input the oblivious output $O_i = O_i(\mathcal{D})$, the oblivious output size $N_i = |O_i|$, and the column c_i, which indicates whether a tuple is part of O_i's true output. Resizer takes also as an input a probability, p_i, sampled from a pre-configured distribution. p_i will be used to determine which filler tuples to retain in the operator output. As output, Resizer returns the shuffled oblivious output O'_i and the shuffled column c'_i, both trimmed to size $S_i = T_i + \eta_i$.

Based on p_i, the Resizer marks tuples to be retained from O_i – note that true tuples are always marked. The marking is stored in column k_i, which is added to O_i. In Figure 4,

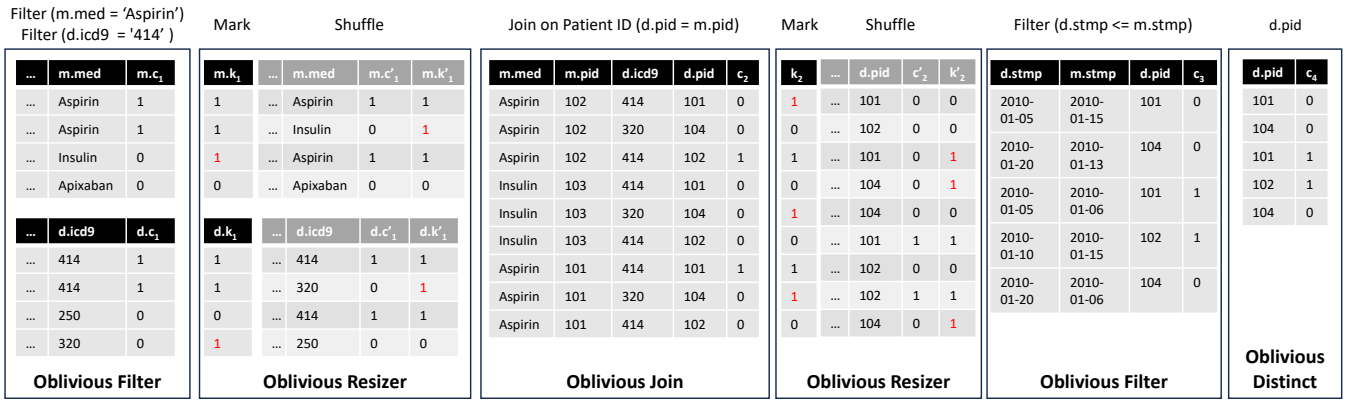


Figure 4. Privacy-preserving query execution with Resizer operator. Retained filler tuples are shown in red.

filler tuples shown in red will be retained in the intermediate results: in columns $d.k_1$ and $m.k_1$ for the two Resizer operators after the Filters, and in k_2 for the one after the join. The output O_i, including all tuples, is then shuffled by k_i to mitigate linkage attacks. Finally, tuples with $k'_i = 0$ are discarded, and O'_i contains only true and filler tuples of size $S_i = T_i + \eta_i$. As discussed before, p_i is sampled obliviously and Resizer operations are executed obliviously; therefore, T_i and p_i remain concealed, and only S_i is revealed.

In the following, we focus on how filler tuples are marked, describing how this step is optimized for efficient MPC execution. The choice of the distribution is deferred to Section 4.3.

4.2.2 Marking Filler Tuples Before Trimming.

The *Mark* step (Figure 3) takes as additional input the p_i sampled from the user-defined distribution attached to the instance of the Resizer, the oblivious output size N_i of the preceding SQL operator O_i, and the column c_i, which identifies true tuples. Mark step outputs a column k_i, indicating tuples to keep from trimming. In the following, we present our approach to efficiently carrying out the marking step.

The marking of tuples to keep is done stochastically: a secret weighted coin (with p_i) is flipped N_i times, with each flip determining whether a tuple is marked for keeping. This method offers a significant advantage in its highly parallelizable nature, as the coin flips for each tuple can occur independently. For each tuple j in the oblivious output O_i of the preceding operator O_i, the following steps are performed (Algorithm 1):

1. A weighted coin is tossed.
2. If the tuple is part of the true output (i.e., $c_i[j] = 1$), its corresponding bit $k_i[j]$ is set to 1, regardless of the coin toss result.
3. Else, i.e., the tuple is not part of the true output ($c_i[j] = 0$), the outcome of the coin toss probabilistically determines its inclusion in the output. A successful toss sets $k_i[j] = 1$, while a failure excludes it by setting $k_i[j] = 0$.

Algorithm 1: Marking of Filler Tuples

Input:

- Sampled success probability p_i.
- Oblivious output size N_i and Column c_i, indicating true tuples, of the preceding SQL operator O_i.

Output: Column k_i indicating tuples to retain.

Step 1: Initialize Data Structures

$k_i \leftarrow []$ // Initialize an empty column.

$rand \leftarrow []$ // Initialize a data structure for sampling random values.

Step 2: Coin Tossing for Random Selection of Filler Tuples

for $j \leftarrow 0$ **to** $N_i - 1$ **do**

  $rand[j] \sim U(0,1)$ // Flip a coin.

  $k_i[j] \leftarrow ((rand[j] < p_i) \vee c_i[j])$ // Compute $k_i[j]$.

Step 3: Return Updated Column

return k_i

With the success probability p_i sampled from the pre-configured distribution $p_i \sim \mathcal{F}(\theta)$, the output after the Resizer satisfies $S_i = T_i + E(\eta_i) \leq N_i$, where $E(\eta_i) = p_i \times (N_i - T_i)$, as dictated by the Binomial distribution.

Tuple-level independence enables natural parallelism, greatly reducing the Resizer's performance overhead. This makes the method well-suited for MPC execution (Section 5.2) by requiring fewer synchronization rounds. Furthermore, its stochastic nature strengthens the privacy guarantees, as discussed in Section 5.3.

Comparison, logic-or, and sampling uniform random numbers of the tuple marking step (Algorithm 1) are directly mapped to their MPC counterparts. The coin flipping, i.e., sampling from the uniform distribution, can be performed in the pre-processing phase of the MPC protocol, and secret shares of the N random numbers are distributed among the parties. In the online phase, the parties compute jointly over the secret shares of both the random values and the inputs.

Since the marking step in Algorithm 1 exhibits natural parallelism, i.e., its loop rounds are independent of each other,

it can benefit from the batching mechanism in MP-SPDZ [1] for improved efficiency. For our setup, we select a batching size of 100, 000.

4.2.3 Shuffle and Trim. A linkage attack occurs when an adversary attempts to exploit information from the intermediate results (in our case, k_i), thereby inferring sensitive connections between secret shares and the corresponding real database values. For instance, if k_i is revealed without shuffling, the attacker can correlate the positions of true negative matches with false positive matches by leveraging memory from a previous execution. If the attacker can identify all true negative matches, they can also determine the true positive data layout.

To prevent linkage attacks [15], shuffling is performed after the Resizer marked the filler tuples (i.e., column k has been created) and before actually trimming the intermediate results. This ensures that no adversary can link secret shares by observing the outputs of different oblivious operators or repeating the same or similar queries multiple rounds. MP-SPDZ [27] implements state-of-the-art shuffling [3, 28] protocol using the Waksman network [38].

In trim, the computing nodes send their shares of k to reconstruct k and discard tuples with $k[j] = 0$.

4.2.4 Complexity Analysis. The computational and communication complexities of the Resizer steps vary by operation in terms of its input size, N, number of oblivious tuples, and M, the tuple width in bytes, and output size S. Tuple marking scales linearly with the number of tuples N, yielding $O(N)$ complexity in both dimensions. Although the algorithm theoretically requires $O(N)$ synchronization rounds, the tuple-level independence of parallel tuple marking reduces the cost through the unrolling and batching mechanism used in MP-SPDZ [27], thereby improving practical performance under MPC. In the RSS protocol, shuffling [29] is not computationally expensive compared to sorting, with a per-party cost of $O(MN \log N)$ for data of length M, and communication cost of $O(N)$. Finally, the trim step combines $O(N)$ and $O(S)$ computation, where S is the size of the trimmed result, but requires only $O(1)$ communication. In our three-party setting, it is expected that the runtime will be dominated by the communication cost.

4.2.5 Placement Strategies. In addition to the flexibility in defining a distribution to sample from for the number of filler tuples to keep, Reflex creates the opportunity to flexibly decide which operators to include a Resizer after. while clearly beneficial in cases when it leads to significant trimming of intermediate results sizes, depending on the expected selectivity and location of the operator in the query tree, including a Resizer can even lead to slowdowns. We explore this question in detail in Section 5.3.

[1] https://mp-spdz.readthedocs.io/en/latest/runtime-options.html

4.3 User-defined Distributions

The role of this distribution is to dictate the amount of filler tuples η_i marked to be kept from trimming. Ideally, η_i should not be greater than $N_i - T_i$, i.e., in the range of $[0, N_i - T_i]$. This is inherently enforced through the loop of Algorithm 1 in Section 4.2.2. By construction, the Resizer will always maintain true tuples, regardless of the outcome of the coin flip. In Reflex, the output of the distribution is actually a probability p_i which determines η_i as the expected value of flipping a p_i weighted coin $N_i - T_i$ times.

The shape and parameters of the distribution strongly affect both performance and privacy. The former is public knowledge, so distributions which produce only a single value (e.g., $p_i = 0.1$) make inferring the true intermediate result size trivial. A distribution biased toward smaller p_i values results, on average, in fewer filler tuples being retained in O_i, which improves performance. Conversely, a distribution with less bias (i.e., closer to uniform) may result in more filler tuples being retained; however, it increases the uncertainty of intermediate result sizes, thereby making it harder for an attacker to recover the true intermediate size T_i, as more observations of S_i values (i.e., more query executions) would be required. In Section 4.4, we present a metric that computes how many observations of S_i are needed by the attacker to recover T_i under specific conditions and show how different distribution parameters affect the number of observations required to recover T_i in Section 5.3.

The objective is to select a distribution shape and parameters that jointly satisfy the performance and privacy requirements of data analysts and data owners. Because the distribution must be determined during query planning, its configuration is limited to the information available at that stage, such as the input size and, when available, operator selectivity. This decision can be further guided by our metric in conjunction with an estimate of the system's performance cost, expressed as the total estimated number of tuples to be processed per query (as shown in Figure 1b).

Overall, the design space of distributions is very large, and it is beyond the scope of this work to automate the process of defining or tuning them for specific workloads. That being said, we provide two example strategies: general distribution and a distribution that can be used to satisfy ϵ-differential privacy (DP) on intermediate results sizes, that is, the truncated Laplace distribution as deployed in Shrinkwrap [5].

4.3.1 Example Distribution: Beta. An intuitive choice for sampling the success probability p_i for the tuple-marking step is the Beta distribution, $Beta(\alpha_i, \beta_i)$. Combined with the Binomial distribution $B(N_i - T_i, p_i)$ representing the tuple marking step, this yields a Beta-Binomial model, $BetaBin(N_i - T_i, \alpha_i, \beta_i)$. This formulation allows p_i to be drawn directly from $[0, 1]$ and then applied as the coin-flip probability over N_i tuples. For each inserted Resizer, the parameters α_i and β_i can be pre-configured independently at the query planning

phase. The expected number of filler tuples $E(\eta_i)$ can be computed as $\frac{\alpha_i}{\alpha_i+\beta_i} \times (N_i - T_i)$.

To improve the overall query performance, the number of (expected) filler tuples $E(\eta_i)$ should be minimized. Since $N_i - T_i$ cannot be controlled at the query planning phase, α_i and β_i should be chosen carefully. The larger the difference between β_i and α_i, with $\beta_i > \alpha_i$, the stronger the tendency toward smaller p_i values and thus smaller $E(\eta_i)$. For instance, a Beta distribution with $\alpha_i = 2$ and $\beta_i = 6$ represents a skewed distribution where the probability p_i is more likely to be closer to 0 than to 1 and the average number of filler tuples $E(\eta_i) = 0.25 \times (N_i - T_i)$.

4.3.2 Example Distribution: DP-based. To provide ϵ-differential privacy guarantees for the intermediate results sizes, distributions such as the truncated Laplace distribution or the exponential distribution can be deployed. For example, in Shrinkwrap [5], truncated Laplace distribution $TLap(\epsilon_i, \delta_i, \Delta c_i)$ is deployed and pre-configured for each operator O_i during the query planning phase:

- Sensitivity (Δc_i) of operator O_i quantifies how much an SQL operator's output can change when a single database record is modified.
- Privacy budget (ϵ_i) reflects the total privacy loss incurred by a mechanism, often expressed using an (ϵ_i, δ_i)-differential privacy guarantee.
- Scale parameter (b_i) is derived from the sensitivity and privacy budget as $b_i = \Delta c_i / \epsilon_i$.
- Location parameter (μ_i) defines the center of the Laplace distribution and is set to a positive value to favor non-negative samples.

Under DP rules, a privacy budget must be maintained, and upon repeated executions of queries, the privacy parameters need to be adjusted accordingly. The effect of this is that, over time, S_i will grow. After exhausting the privacy budget, operators must behave fully oblivious.

DP-based distributions can be deployed with our *tuple marking* approach by computing p_i from η_i: This is done by trimming η_i at runtime to $min(N_i - T_i, \eta_i)$ and dividing by $N_i - T_i$ to ensure the success probability of $p_i \in [0, 1]$. The expected number of filler tuples value $E(\eta_i)$ is again $p_i \times (N_i - T_i)$.

4.4 Proposed Metric: Rounds to Recover

To better understand the effect of Reflex on the privacy of intermediate results *sizes*, we define a conservative metric that computes the *number of equivalent repetitions of an operator required to recover the operator's true result size T* (in short, Rounds to Recover). An equivalent repetition of an operator's execution occurs when, even in different queries, the operator is executed with the same input and yields the same true intermediate result size. This definition could be further refined to account for adversarially crafted queries designed to exploit repetitions of an operator, which take slightly different inputs and produce slightly different output sizes. However, such considerations fall outside the scope of this work, as we focus on enterprise settings where data analysts are not assumed to act maliciously.

The metric quantifies how often a pre-configured Resizer placed after some operator can execute on a dataset $\mathcal{D}$ before the size of the true intermediate result of the operator has to be considered as revealed. Note that this metric does not quantify the privacy loss of individual tuples in the intermediate results due to deploying Reflex. Assuming a passive attacker (see Section 3) collects r observations (samples) of the operator O trimmed output size over r queries: $S_1, S_2 \ldots S_r$, where $k = 0 \ldots r-1$ and each $S_k = T + \eta_k$ and η_k is the amount of filler tuples that is *independently sampled* from a distribution $\mathcal{F}(\mu_\eta, \sigma_\eta^2)$ with a *finite* mean value μ_η and a *finite* variance value σ_η^2, and added to T, our secret value. We assume the distribution characteristics to be known.

The trimmed output size S follows the same distribution of η with a mean value $\mu_s = \mu_\eta + T$ and a variance value $\sigma_s^2 = \sigma_\eta^2$.

After sufficient observations, the attacker can compute the average of observations $\bar{S} = \frac{1}{r}\sum_{k=1}^{r} S_k$ to approximate/estimate the mean value μ_s. That is:

$$\bar{S} = T + \frac{\sum_{k=1}^{r} \eta_k}{r} = T + \bar{\eta} \approx T + \mu_\eta.$$

Hence, the attacker can estimate the true output size: $T \approx \bar{S} - \mu_\eta$. Based on the Central Limit Theorem (CLT), $\bar{S}$ follows a normal distribution, with a mean value μ_s and a variance value σ_s^2/r, for large enough r. Using confidence intervals $P(|\bar{S} - \mu_s| \leq \text{err}) \geq \alpha$, we can set an error margin, $\text{err} \leq z_{\alpha/2} \times \frac{\sigma_s}{\sqrt{r}}$, where ($z_{\alpha/2}$ is the z-score corresponding to the desired confidence level. For example, for $\alpha = 99.9\%$, ($z_{\alpha/2} = 3.291$).

Therefore, the number of rounds r, representing our RtR metric, can be computed as

$$RtR \geq z_{\alpha/2}^2 \times \frac{\sigma_s^2}{\text{err}^2} \tag{1}$$

5 Evaluation

We describe our setup and objectives in Section 5.1. Next, we analyze the cost of Resizer and compare it to SQL operators in Section 5.2, examine the performance–privacy trade-off in Section 5.3, evaluate the runtime of HealthLnk and TPC-H queries, and finally compare Reflex to related work in Section 5.4.

5.1 Setup and Goals

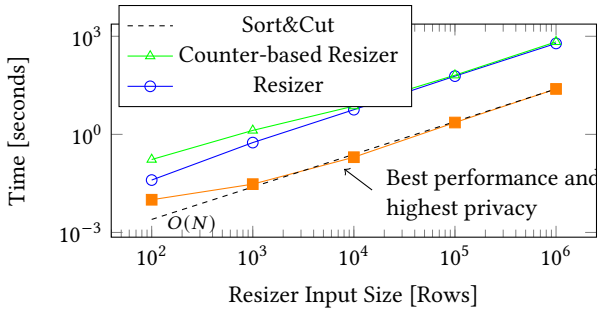
We evaluate Reflex on three physical machines, each equipped with 256 GBx2 of RAM and two with Intel(R) Xeon(R) Gold 5220 CPU @ 2.20 GHz, another one with Intel(R) Xeon(R) Gold 5120 CPU @ 2.20 GHz, providing a robust and scalable environment for secure computation. Data for the queries

is loaded locally from each machine, and an additional external client starts the execution. We evaluated each query individually over three computing nodes in a local area network (LAN) environment with an average round-trip time (RTT) latency of 0.25 ms, averaging the results across five independent runs. Overall, our experimental evaluation sets out to provide answers to three sets of questions:

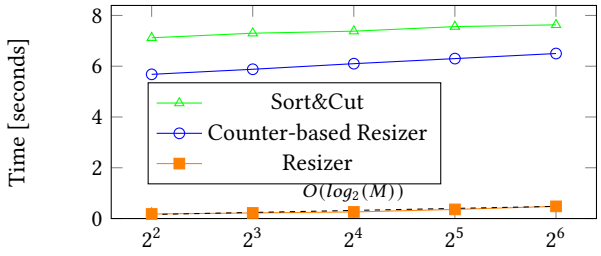
Q_1: *How expensive is a Resizer compared to oblivious SQL operators? How does its runtime scale with more and wider tuples?* – We provide answers in Section 5.2 and show that a Resizer is comparable in runtime cost to the operators and gracefully scales with increasing data sizes.

Q_2: *What is the practical implication of the performance–privacy trade-off when revealing intermediate result sizes? How easily can an attacker learn the true intermediate result sizes?* – This trade-off is explored in Section 5.3, where we show that there is no silver bullet – but there is a large space to explore and, with Reflex, future query optimizers will be able to decide where to use how much protection in order to reach a performance target.

Q_3: *How much faster are queries when Reflex is used to trim the intermediate result sizes? Are the speedups similar to expectations based on related works?* – We explore this in Section 5.4 using HealthLnk and TPC-H queries. We demonstrate that Reflex enables orders of magnitude faster query execution than fully oblivious and other related work approaches, while maintaining at least the same level of security as other related work that discloses trimmed intermediate result sizes.



(a) Runtime with increasing row count and fixed row width of 16 bytes (plaintext)



(b) Runtime with increasing row width and fixed row count of 10000

Figure 5. Resizer demonstrates linear scalability with the number of rows and logarithmic scalability with the number of columns. It outperforms the counter-based version and sort&cut by more than an order of magnitude.

5.2 Runtime Cost of Resizer

Resizer vs. other approaches. In Figure 5, In Figure 5, we examine how the runtime of Resizer scales with increasing input size, measured by the number and width of tuples. We compare Resizer against two alternatives: (1) A counter-based Resizer that has the same steps as Resizer (mark, shuffle, and trim). For this approach, the number of filler tuples to retain is sampled directly, $\eta_i \sim \mathcal{F}(\theta)$, and η_i is assumed to be some value between $[0, N_i]$. In counter-based Resizer 's mark step, the tuples are processed sequentially in order. For each tuple j of O_i, the following is executed: If the counter has not reached the η_i value, or if the tuple belongs to the true output (i.e., $c_i[j] = 1$), its corresponding bit $k_i[j]$ is set to 1. Otherwise, the tuple is excluded, and $k_i[j]$ is set to 0. This step is followed by shuffling and trimming.

(2) A sort&cut approach as presented in [5]. Here, the tuples of O_i are first sorted in descending order based on column c_i, after which some filler tuples are trimmed from the bottom. The number of filler tuples retained in O_i is determined by sampling the truncated Laplace distribution.

Note that the selectivity and the number of filler tuples have no effect on the Resizer's runtime since all tuples must be processed by the Resizer. Figure 5a shows the behavior of Resizer with increasing number of tuples, where each tuple has a fixed width of 4 columns, and each column is 4 bytes in plaintext, resulting in a tuple width of $4 \times 4 = 16$ bytes. We compare the runtime of Resizer with the counter-based Resizer and our implementation of the sort&cut method [5] in MP-SPDZ.

As expected, Resizer runtime scales linearly with the number of rows in its input, still an order of magnitude faster than the counter-based Resizer. Although the counter-based Resizer is simple to implement, its inherently sequential nature renders it inefficient for MPC execution. It requires a secure counter to be checked in each iteration, to determine whether the count of filler tuples marked so far has reached the limit η_i, thereby creating a loop dependency. In contrast, for the Resizer each iteration is independent, enabling parallelism and thus more efficient execution.

Furthermore, as illustrated in Figure 5a, both Resizer and counter-based Resizer outperform the sorting-based solution. This improvement arises because secure shuffling is less computationally expensive than secure sorting [5].

We also conducted an experiment to show that the width of tuples plays a less critical role in the runtime of the Resizer. We fixed the number of tuples at $N = 10,000$. As the table width increases from 2 to 64 columns (with 10,000

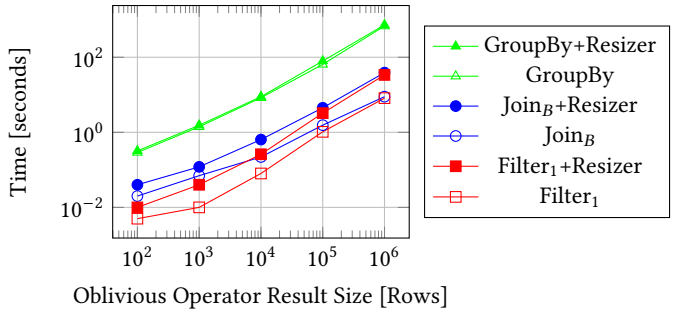


Figure 6. Adding Resizer after an operator increases runtime, but reducing the output size will speed up the next operator.

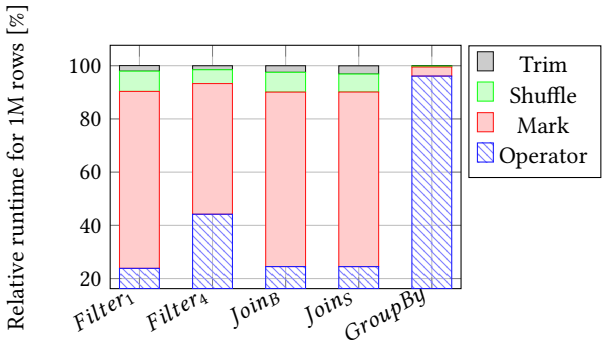


Figure 7. Depending on the oblivious operator, Resizer could be relatively cheap compared to the operator itself. Filter$_1$ is a filter with 1 equality condition, Filter$_4$ is a filter with 4 equality conditions, Join$_B$ has two balanced tables, each with a size equal to the square root of the output size, as input, and Join$_S$ has two unbalanced tables as input.

rows fixed), the Resizer runtime rises gradually, exhibiting a logarithmic growth. Tuple width does not affect the mark or trim steps of the Resizer, both steps operate purely row-wise, acting on the c_i and k_i columns. However, the shuffle step involves a copy operation, which can be slightly impacted by the width of the tuples being copied. As shown in Figure 5b, increasing the number of columns results in only a sublinear increase in cost. Overall, the runtime of Resizer is both the lowest compared to the other variants.

Combining Oblivious Operators with Resizer. In this experiment, we compare the runtime overhead of Resizer to other database operators under different intermediate result sizes (before trimming). In the context of a single operator followed by a Resizer, the number of rows that are being trimmed away plays no role in the runtime of either the operator or the Resizer. Its effect is only visible for the next operator, with less input data to process. For this reason, in Figure 6 and Figure 7, we do not show operator selectivity or the count of filler tuples.

Figure 6 shows the runtime of oblivious operators with and without a Resizer. We focused here on (1) a filter with one equality condition (Filter$_1$), (2) a join that has two balanced tables as input, each has a size equal to the square root of the output size (Join$_B$), and (3) a group by operator that groups based on one column. Thanks to the linear runtime of Resizer, which is independent of the type of operator executing before it, the results in Figure 6 are predictable, as performance does not degrade significantly.

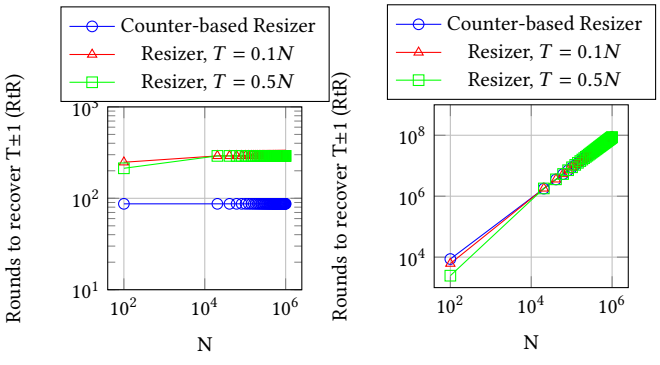
In Figure 7, we show how expensive a Resizer is relative to the actual operators, with a fixed 1M intermediate result size (i.e., the oblivious output of the operator and the input to the Resizer has 1M rows). We compared Filter with one equality condition (Filter$_1$) and four equality conditions (Filter$_4$), a balanced join (Join$_B$) with two input tables of size $sqrt(N)$, an unbalanced join (Join$_S$) of $1 : N$, and a group by. As seen in Figure 7, we tested each step of Resizer separately. The mark step of Resizer is more expensive than Filter$_1$ but cheaper than the GroupBy. This is because for each tuple, the Resizer needs to conduct an online comparison and a logical OR gate over secret shares. However, Filter$_1$ only requires one equality check. Similarly, if the operator is more complex, such as GroupBy (which includes sorting as a pre-operation), the mark step of Resizer will seem relatively cheap. The shuffle step is performed in constant rounds, with the final trim step being cheaper than the previous operations.

5.3 Privacy vs. Performance

In this evaluation, we use the Rounds to Recover (RtR) in Equation (1) derived in Section 4.4. RtR shows how often an attacker needs to observe the MPC execution of equivalent repetitions to recover the true intermediate size T_i of an operator O_i, within a given error margin when Reflex is deployed. Recall $RtR \geq z^2_{\alpha/2} \times \sigma_s^2/err^2$. In our experiments, we fix $z_{\alpha/2} = 3.291$, i.e., to compute RtR that achieves a confidence level of 99.9%, and examine the effect of different distributions, represented by their variances and error margins. Furthermore, we examine the performance–privacy trade-off.

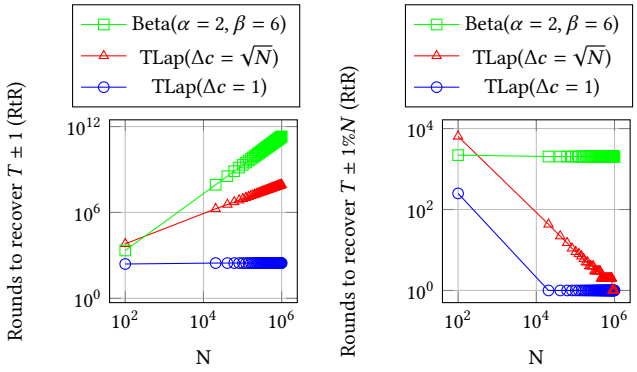
Resizer vs. Counter-based Resizer. In this experiment, we evaluate the impact of coin tossing-based mark step on RtR. We compare the coin-tossing-based mark with the counter-based mark step Section 5.2. We use the ϵ-differential truncated Laplace distribution $TLap(\epsilon = 0.5, \delta = 0.00005)$ on the range $[0, \infty)$ [5] and fix the error margin to 1 – in other words, the attacker "wins" if $T \pm 1$ can be determined with 99.9% confidence level.

When using the Counter-based resizer, η will be sampled directly from $TLap(\epsilon = 0.5, \delta = 0.00005)$. In case of coin-tossing, p is computed as $min(eta, N)/N$ (with η sampled as mentioned above). Using the law of variance, we compute the variance of the combined truncated Laplace and Binomial distributions for two values of T ($T = 0.1 \times N$ and $T = 0.5 \times N$). In addition, we test under different sensitivities ($\Delta c = 1$ and



(a) RtR for TLap($\Delta c = 1$) **(b)** RtR for TLap($\Delta c = \sqrt{N}$)

Figure 8. Coin tossing-based Resizer (in red and green) mostly outperforms counter-based Resizer with $T = 0.1N$ (in blue). More observations of the operator's noisy output size S are needed to recover the true output size T with an error margin of 1 tuple. Both figures deploy a truncated Laplace distribution from [5] $TLap(\epsilon = 0.5, \delta = 0.00005)$. The left-side figure has a narrower distribution with $\Delta c = 1$ and a scale $b = \frac{1}{\epsilon} = 2$, whereas the right-side figure has a wider distribution with $\Delta c = \sqrt{N}$ a scale $b = \frac{\sqrt{N}}{\epsilon} = 2\sqrt{N}$.



(a) RtR with $err = 1$ **(b)** RtR with $err = 1\%N$

Figure 9. RtR highly depends on the error margin accepted by the attacker. Combining Resizer with Beta distribution (green plots) results in more observations required to recover T even with the attacker tolerating accuracy (by allowing a higher error margin in Equation (1)). Note that all plots use $T = 0.05N$. For a narrow truncated Laplace distribution ($\Delta c = 1$), RtR = 1, i.e., one observation, for small T values.

$\Delta c = \sqrt{N}$), which affect the shape of the distribution (narrow or wide).

For the TLap distribution with a low scale $b = 2$ ($\Delta c = 1$), Figure 8a demonstrates that the coin tossing-based mark results in more rounds RtR than the counter-based mark, even for larger T values. Counter-based mark can only achieve comparable privacy guarantees when combined with a TLap distribution of a higher scale $b = 2\sqrt{N}$ ($\Delta c = \sqrt{N}$) as shown in Figure 8b.

Impact of Distribution. In this experiment, we use Resizer with two different distributions: the truncated Laplace distribution, as defined above, and the Beta distribution $B(\alpha = 2, \beta = 6$, described in Section 4.3 and utilize an error margin of $T \pm 1$, our results in Figure 9a indicate that Beta-binomial distribution achieves more RtR compared to truncated Laplace, even with high scale value $b = 2\sqrt{N}$.

Larger values of r affect performance. To analyze the trade-offs, we evaluate the runtime of Resizer with the different distributions: $TLap(\epsilon = 0.5, \delta = 0.00005, \Delta c = 1000)$ and $Beta(\alpha = 2, \beta = 6)$ under the same workload as in Figure 10 (right-side), with $N = 1M$ and $T = 10\%N$, for the operation sequence $\text{Join}_B \rightarrow$ Resizer$\rightarrow$ OrderBy. Resizer configured with $TLap(0.5, 0.00005, 1000)$, resulting in an average count of filler tuples of approx. $2\%N$, achieved a runtime of 104s. In comparison, using $Beta(2, 6)$, which introduces an average count of filler tuples of $25\%(N - T) = 22.5\%N$, resulted in a runtime of 236s.

We analyze the impact of the error margin on the number of rounds an attacker needs for the two distributions when combined Resizer. Relaxing the error margin allows for an informal examination of the potential extent of information leakage on T. Specifically, we investigate how many rounds an attacker would need to guess T within a specified range relative to N. Our findings in Figure 9b demonstrate that relaxing the accuracy requirement for T can dramatically reduce the number of rounds. For instance, with small T values (e.g., $T = 5\% \times N$), using a narrow distribution ($b = 2$) and permitting a recovery accuracy of $T \pm 1\% \times N$ reduces the number of rounds to $RtR = 1$ (i.e., the attacker needs to observe only a single repetition to discover $T = 5\% \times N$). In contrast, wider distributions can still offer acceptable guarantees, but the expected runtime will be significantly larger, as fewer tuples are trimmed on average. We believe the RtR metric is a crucial step towards enabling the query optimizer to consider the privacy implications of different distributions and balance them with their performance implications.

Resizer Placement – Micro-benchmarks. Based on the results in Figure 7, it is clear that a Resizer incurs an additional runtime cost, which must be balanced against the benefits of data reduction as the query tree is traversed. In the previous experiment, we placed a Resizer after each internal operator, but in the future, a query optimizer should decide on a per-operator basis, based on its expected selectivity and the magnitude of expected filler tuples, whether a Resizer is worth inserting or not. To show what *cost functions* an optimizer might use, we depict the performance of two recurring operator combinations: a join followed by a filter ($\text{Join}_B \rightarrow \text{Filter}_1$), and a join followed by an order by (Join_B $\rightarrow$ OrderBy). We test the overall runtime with and without a Resizer after the join in both combinations.

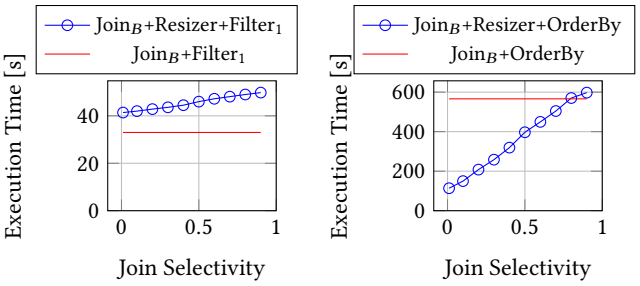


Figure 10. The left figure shows that inserting Resizer (that adds 10% of the FO-Join output size) between Join and Filter is not beneficial, whereas the right figure shows that between Join and OrderBy it will be almost always beneficial.

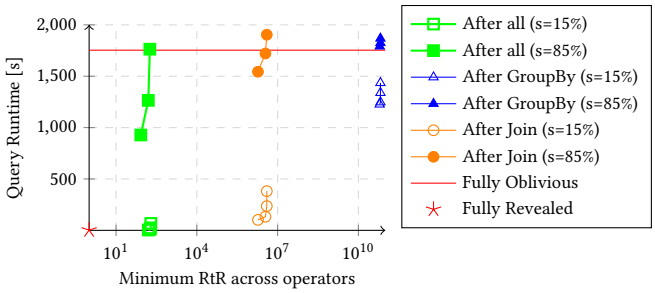


Figure 11. Different Resizer placement rules for modified TPC-H Q3, with average SQL operator selectivity of s=15%, respectively s=85%, impacts RtR (depicted: minimum RtR across operators in the query). Greedily adding a Resizer after each oblivious operator isn't always optimal. The Resizer integration method impacts both performance and privacy.

In Figure 10, we show the performance of the query snippets, assuming different selectivity of the join. Interestingly, in Figure 10 (left-side), the runtime with Resizer is always higher than without it. Hence, placing the Resizer before the filter, with the filter as the last operator in the query, would actually slow it down. For the case in Figure 10 (right-side), the opposite is true, and inserting the Resizer speeds up execution except for selectivity > 85%.

In fully oblivious execution, the query plans have little space for optimization. With Reflex, query optimization is possible. As this example shows, it is straightforward to construct cost functions for future query optimizers to inform their decisions.

Effect of Resizer Placement in a TPC-H Query. In this experiment, we evaluate the impact of different placement rules for Resizer on both the runtime and the Rounds to Recovery (RtR). The three rules are: insert a Resizer after all intermediate operators, only after joins, or only after GroupBys (plotted as lines with marks on Figure 11). We assume the Resizers are configured to receive p values using the Beta distribution. For datasets with 15% selectivity, we used four configurations: 5%, 10%, 20%, and 30% of input size, as the average count of filler tuples. For 85% selectivity, we used: 5%, 10%, and 15% of the input size as the average count of filler tuples. Although the relative RtR behavior is valid across distributions, it is essential to note that the choice of user-defined distribution has a significant impact on the absolute value and, consequently, the real-world security of the system.

We evaluate the runtime and the RtR metric of these placements on TPC-H Q3, with data modified such that all operators have either around 15% or 85% selectivity – to show the selectivity spectrum on the same query tree. The reported RtR value reflects the minimum among RtR values of all operators in Q3 (e.g., the weakest link in the query plan).

The red star on Figure 11 denotes the execution of Q3 without any Resizer insertions and with all intermediate result sizes are fully revealed. This configuration yields the best performance but no protection, as a single observation is sufficient to recover the true sizes of the intermediate results of all operators (i.e., $RtR = 1$). In contrast, the red bar represents the execution time of Q3 in fully oblivious mode, where no Resizer is used. This setup offers the highest level of privacy (i.e., $RtR \to \infty$), and is depicted as a bar to indicate that any strategy with a higher execution time (i.e., that crosses this bar) is considered impractical.

Key insights from this experiment can be summarized as follows: First, inserting Resizer after every operator improves runtime but severely reduces privacy (low RtR), as fewer rounds are needed to infer the true intermediate result size of the weakest operator in the query plan. Therefore, a more targeted placement of Resizer can yield comparable performance gains while achieving higher RtR. For example, the *After Join* rule has runtime similar to the *After all* rule, but with better RtR.

Second, placing Resizer after GroupBy operators high up the query tree results in higher RtR but worse performance compared to placing it after Join operators. This suggests that placing Resizer earlier in the query plan is perhaps a generally better choice.

Third, as the average selectivity of SQL operators increases, the effectiveness of Resizer diminishes: execution time worsens while RtR remains similar. For example, the *After GroupBy* and *After Join* rules exhibit worse runtimes than the fully oblivious baseline under high selectivity.

Finally, the choice of distribution, i.e., the number of filler tuples added to the true intermediate result, clearly has a significant impact on performance. Adding more filler tuples has a super-linear increase in runtime. However, in most cases, it does not always lead to proportionally better RtR. For all plotted placement rules, beyond a certain point, retaining more filler tuples no longer meaningfully increases the number of rounds required to infer true intermediate result sizes.

System	Security	MPC Framework (Domain)	IRS Protection (Operators)	Configurable Protection	Select Algorithm	Join Algorithm	Group By Algorithm
SMCQL [4]	Semi-Honest	Oblivm [32] (Boolean)	Fully Oblivious (All)	○	Sequential	NLJ	Sort-based
Shrinkwrap [5]	Semi-Honest	EMP-toolkit [43] (Boolean)	DP-based Trimming (All)	◒	Sequential	NLJ	Sort-based
Secretflow [17]	Semi-Honest	SCQL [1] (Boolean)	Fully Oblivious or Revealed (Join, Group by)	◒	Sequential	PSI-Join	Sort-based
Secrecy [31]	Semi-Honest	Self-implemented (Hybrid)	Fully Oblivious (All)	○	Sequential	NLJ	Sort-based
Reflex (this work)	Semi-Honest	MP-SPDZ [27] (Hybrid)	Flexible Trimming (All)	●	Sequential	NLJ	Sort-based

Table 1. Comparison to related work. Reflex is the only system offering a flexible and efficient Resizer, enabling users to balance privacy and performance.

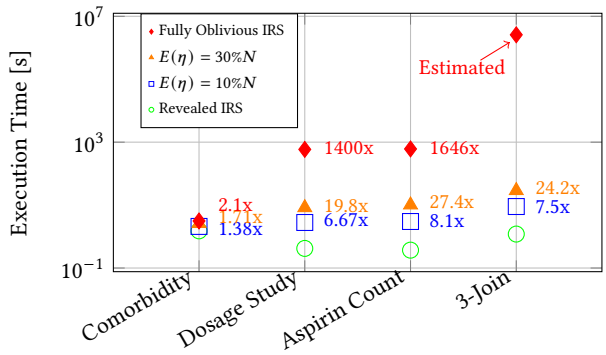


Figure 12. We apply Resizer to the HealthLnK queries (synthetic data with table sizes of 1000 and fixed selectivity of 10% per operator) and show that, when trimming filler tuples, performance is dramatically increased. If no fillers are trimmed, query runtimes increase by more than 1000x.

5.4 Analytical Queries from Related Work

HealthLnk Queries in Reflex. We investigate the runtime of HealthLnk queries [35] used in related work [4–6], that are indicative of clinical data research methodologies [24]. In Figure 12 we present the execution times of the Comorbidity Study, Dosage Study, Aspirin Count, and 3-join queries. Unless otherwise specified, each table has been populated with $N = 1000$ synthetic tuples, in a way that filter and join selectivities are $10\%N$ per operator.

Figure 12 compares the runtimes (in seconds) of Reflex in the following settings: (i) fully oblivious execution without intermediate trimming (red diamonds), resulting in fully oblivious intermediate result sizes (IRS), (ii) Reflex using Resizer operators with a probability distribution that in expectation keeps $\eta = 30\%N$ filler tuples (orange triangles), (iii) Reflex with $\eta = 10\%N$ (blue squares), and (iv) oblivious execution of operators but trimming all filler tuples (green circles), effectively revealing the IRS. We placed a Resizer operator after each operator in the query, except for the last operator.

The results show that reducing IRS leads to significantly reduced runtime for queries with joins (Dosage Study, Aspirin Count, 3-Join). Since the Comorbidity query does not involve a join operation, it is less affected by ballooning data sizes, benefiting only modestly from trimming. Note that the runtime of the 3-Join query under a fully oblivious setting is estimated (it could not be executed on our platform due to memory limitations). We estimate the runtime by measuring its runtime with smaller table sizes and extrapolating using the trend shown in Figure 1a. Overall, depending on the complexity of the queries to be executed – and in particular the number of joins – Reflex can substantially boost performance when trimming away fillers.

Comparison to Related Work. Table 1 compares Reflex with representative semi-honest analytics systems that use MPC. The table shows i) choice of MPC Framework and computational domain, which has an important effect on overall performance, ii) a summary of how these frameworks protect intermediate result size and whether they expose some user-facing parameters for relaxing these protections, and iii) the underlying algorithms used for filter, join, and group by. Due to the numerous design differences across frameworks and variations in implementation, it is relatively challenging to provide an apples-to-apples comparison of these systems. The goal of this table is to provide an overview and show that Reflex is based on a comparable foundation to the state of the art. In terms of baseline performance of the MPC frameworks, given the analytics use-case, those that operate in an arithmetic or hybrid model are expected to be significantly faster than those using solely binary: SMCQL [4] and Shrinkwrap [5] rely on Boolean-circuit frameworks (Oblivm [32] and EMP-tookit [43]). SecretFlow [17] also uses a Boolean domain in SCQL [1]. Secrecy and Reflex execute in hybrid mode, combining arithmetic and Boolean domains for improved efficiency.

In terms of protecting the intermediate result size, SMCQL [4] and Secrecy [31] enforce full obliviousness, incurring a high cost on operations such as joins and group by. Shrinkwrap [5] introduces a differentially private (DP) mechanism to reduce overhead. The user can influence the trimming behavior indirectly through the privacy budget and other parameters of the DP mechanism. Secretflow [17] can operate in two modes: either keeping IRS oblivious, without performance benefits, or exposing it fully. As the table shows, Secretflow offers additional security relaxations that users can choose, allowing, for instance, parties to learn something about the contents of a join result. However, this results in semi-oblivious execution of core operators, which no other system compromises on. Reflex is the only system with flexible IRS protection based on user-defined distributions. These distributions could even be used to provide the DP guarantees from related work.

Query processing happens in a similar fashion in all systems: they employ sequential selection and sort-based group-by [20]. For joins, most use a nested-loop join (NLJ), which

iteratively compares each pair of tuples from the input relations, whereas SecretFlow [17] adopts a PSI-join, i.e., a join based on private set intersection [25] that securely identifies matching join keys between two datasets – this algorithm, while only usable for key-to-key joins, has the better asymptotic complexity than NLJ. Overall, Reflex uses algorithms similar to the state of the art and leverages the maturity and efficiency of the underlying MP-SPDZ framework.

6 Summary

We presented Reflex, an efficient and flexible approach for trimming oblivious intermediate results in MPC-based query execution.

At the core of Reflex is the Resizer that can be inserted after any oblivious operator to trim its intermediate result size, without requiring query rewrites or changes to upstream operators. The decision of how many filler tuples to trim is taken using user-defined probability distributions. These allow capturing different points in the trade-off space. To enable a statistically-grounded comparison between distributions, we introduce the Round to Recover (RtR) metric, which applies to any distribution and helps planners select the most secure strategy within a given time budget.

We evaluated Reflex using the queries from related work [4–6] and achieved runtime reduction thanks to trimmed intermediate results in similar orders of magnitude to related work. Generally, Reflex is even faster thanks to its efficient implementation in MP-SPDZ. Reflex paves the way for future query optimizers that consider both privacy and performance, as it allows future query optimizers to compile SQL queries with Resizer included, and, from our results, cost models can be derived on the effect of filler tuples on operator runtime.

Artifacts

The source code and data used in this paper are available at: https://github.com/DataManagementLab/reflex-smpc-analytics

References

[1] 2024. SecretFlow-SCQL: A Secure Collaborative Query pLatform. *Published in Proc. VLDB Endow.* 17, 12 (2024), 3987–4000. https://github.com/secretflow/scql

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 805–817.

[3] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 125–138. https://doi.org/10.1145/3548606.3560691

[4] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2016. SMCQL: Secure Querying for Federated Databases.

[5] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018).

[6] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. Saqe: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2691–2705.

[7] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. 2023. HE3DB: An Efficient and Elastic Encrypted Database Via Arithmetic-And-Logic Fully Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2930–2944.

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure:SGX cache attacks are practical. In *11th USENIX workshop on offensive technologies (WOOT 17)*.

[9] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards practical oblivious join. In *Proceedings of the 2022 International Conference on Management of Data*. 803–817.

[10] European Commission. [n.d.]. https://commission.europa.eu/law/law-topic/data-protection/rules-business-and-organisations/obligations/what-does-data-protection-design-and-default-mean_en.

[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf

[12] Jesse De Meulemeester, David Oswald, Ingrid Verbauwhede, and Jo Van Bulck. 2026. Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing. In *47th IEEE Symposium on Security and Privacy (S&P)*.

[13] Tassos Dimitriou and Antonis Michalas. 2014. Multi-party trust computation in decentralized environments in the presence of malicious adversaries. *Ad Hoc Networks* 15 (2014), 53–66.

[14] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved primitives for MPC over mixed arithmetic-binary circuits. In *Annual international cryptology conference*. Springer, 823–852.

[15] Saba Eskandarian and Dan Boneh. 2021. Clarion: Anonymous communication from multiparty shuffling protocols. *Cryptology ePrint Archive* (2021).

[16] Saba Eskandarian and Matei Zaharia. 2017. Oblidb: Oblivious query processing for secure databases. *arXiv preprint arXiv:1710.00458* (2017).

[17] Wenjing Fang, Shunde Cao, Guojin Hua, Junming Ma, Yongqiang Yu, Qunshan Huang, Jun Feng, Jin Tan, Xiaopeng Zan, Pu Duan, et al. 2024. SecretFlow-SCQL: A Secure Collaborative Query pLatform. (2024).

[18] Jun Furukawa and Yehuda Lindell. 2019. Two-Thirds Honest-Majority MPC for Malicious Adversaries at Almost the Cost of Semi-Honest. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery.

[19] General Data Protection Regulation GDPR. 2016. General data protection regulation. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC* (2016).

[20] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.

[21] Antonio Guimarães, Edson Borin, and Diego F Aranha. 2021. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 229–253.

[22] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable collaborative analytics system on private database with malicious security. In *2022 IEEE 38th International Conference*

on Data Engineering (ICDE). IEEE, 1740–1753.

[23] Xi He, Jennie Rogers, Johes Bater, Ashwin Machanavajjhala, Chenghong Wang, and Xiao Wang. 2021. Practical security and privacy for database systems. In *Proceedings of the 2021 International Conference on Management of Data*. 2839–2845.

[24] Adrian F Hernandez, Rachael L Fleurence, and Russell L Rothman. 2015. The ADAPTABLE Trial and PCORnet: shining light on a new research paradigm. , 635–636 pages.

[25] Bernardo A Huberman, Matt Franklin, and Tad Hogg. 1999. Enhancing privacy and trust in electronic communities. In *Proceedings of the 1st ACM conference on Electronic commerce*. 78–86.

[26] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* 13 (2016), 12.

[27] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.

[28] Marcel Keller and Peter Scholl. 2014. Efficient, oblivious data structures for MPC. In *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*. Springer, 506–525.

[29] Marcel Keller and Peter Scholl. 2014. Efficient, oblivious data structures for MPC. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 506–525.

[30] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. 557–574.

[31] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1031–1056.

[32] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 359–376.

[33] Adrian Lutsch, Muhammad El-Hindi, Matthias Heinrich, Daniel Ritter, Zsolt István, and Carsten Binnig. 2025. Benchmarking Analytical Query Processing in Intel SGXv2. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*, Alkis Simitsis, Bettina Kemme, Anna Queralt, Oscar Romero, and Petar Jovanovic (Eds.). OpenProceedings.org, 516–528. https://doi.org/10.48786/EDBT.2025.41

[34] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/2487726.2488368

[35] Patient-Centered Outcomes Research Institute (PCORI). 2015. Exchanging de-identified data between hospitals for city-wide health analysis in the Chicago Area HealthLNK data repository (HDR).

[36] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.

[37] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. 2020. Cryptε: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 603–619.

[38] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2023. Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3328–3342.

[39] Alex Seto, Oytun Kuday Duran, Samy Amer, Jalen Chuang, Stephan van Schaik, Daniel Genkin, and Christina Garman. 2025. WireTap: Breaking Server SGX via DRAM Bus Interposition. In *2025 SIGSAC Conference on Computer and Communications Security (CCS '25)*. Association for Computing Machinery. https://wiretap.fail

[40] Donghyun Sohn, Kelly Jiang, Nicolas Hammer, and Jennie Rogers. 2025. Alchemy: A Query Optimization Framework for Oblivious SQL. *Proceedings of the VLDB Endowment* 18, 9 (2025), 3021–3034.

[41] Sven Trieflinger. 2020. *Trustworthy computing – data sovereignty while connected.* https://www.bosch.com/research/research-fields/digitalization-and-connectivity/research-on-security-and-privacy/trustworthy-computing-data-sovereignty-while-connected/

[42] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.

[43] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit.

[44] Rishabh Poddar Sukrit Kalra Avishay Yanai, Ryan Deng, and Raluca Ada Popa Joseph M Hellerstein. 2021. Senate: A maliciously-secure MPC platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21), Vancouver, BC*.