SECRECY: Secure collaborative analytics in untrusted clouds

John Liagouris *Vasiliki Kalavri* *Muhammad Faisal* *Mayank Varia*
Boston University
{liagos, vkalavri, mfaisal, varia}@bu.edu

Abstract

We present SECRECY, a system for privacy-preserving collaborative analytics as a service. SECRECY allows multiple data holders to contribute their data towards a joint analysis in the cloud, while keeping the data siloed even from the cloud providers. At the same time, it enables cloud providers to offer their services to clients who would have otherwise refused to perform a computation altogether or insisted that it be done on private infrastructure. SECRECY ensures no information leakage and provides provable security guarantees by employing cryptographically secure Multi-Party Computation (MPC).

In SECRECY we take a novel approach to optimizing MPC execution by co-designing multiple layers of the system stack and exposing the MPC costs to the query engine. To achieve practical performance, SECRECY applies physical optimizations that amortize the inherent MPC overheads along with logical optimizations that dramatically reduce the computation, communication, and space requirements during query execution. Our multi-cloud experiments demonstrate that SECRECY improves query performance by over 1000× compared to existing approaches and computes complex analytics on millions of data records with modest use of resources.

1 Introduction

Secure collaborative analytics [20,26,30,47,97] is a family of emerging applications, where multiple data holders are willing to allow certain computations on their collective private data (e.g., for profit, social good, improved services, etc.), provided that the data remain siloed from untrusted entities. For instance, some companies would agree to participate in a gender wage gap study [32] but only if no employee wages are revealed to other companies, as they may lose their competitive advantage. Similarly, researchers from different medical institutions may conduct a large-scale study on the union of their patient records, provided that the data analysis is end-to-end compliant with privacy regulations [2, 3]. Another example is private advertising: web users may subscribe to recommendations based on collaborative filtering as long as their online activity remains hidden from the service provider [85].

To realize the above use cases, we need systems capable of extracting value from sensitive or proprietary data, while protecting the data from untrusted or unauthorized entities. We identify four major requirements for such systems. First, they must ensure no information leakage so that they reveal nothing except the output of the computation the data holders have agreed on. At the same time, they must guarantee security in the absence of trusted resources, as the data holders may lack the expertise or infrastructure needed for secure computation and may need to outsource the analysis to untrusted third parties [45]. Another requirement is to support complex analytics beyond simple statistics, such as relational queries on multiple tables [95]. Lastly, while queries in these use cases are non-interactive, they must complete in reasonable time, e.g., within a few hours.

Enabling secure outsourced analytics with practical performance has been a long-standing research challenge [13]. So far, there exist three general approaches to secure computation with no leakage. The first one is Fully Homomorphic Encryption (FHE) [57] that provides "ideal" security by enabling computation directly on encrypted data. Although there are many implementations that support simple functions [4, 6, 8, 21], FHE is still prohibitively slow for the analytics we consider in this work. A more practical approach is to use secure hardware solutions, like Intel's SGX, which have been proposed as a faster alternative to cryptography but do not provide provable security [33, 74]. A third promising approach is cryptographically secure Multi-Party Computation (MPC) [77]. MPC refers to a family of cryptographic protocols that enable mutually distrusting parties to jointly compute functions on secret (encoded) data without relying on any single trusted entity. MPC is generally faster than FHE-based approaches but still challenging to scale to inputs with more than a few thousand records [22, 45, 73, 95, 105].

Recently, systems like Conclave [105], SMCQL [22], Senate [95], and others [23, 24, 111] have made MPC more accessible to data analysts by providing relational interfaces

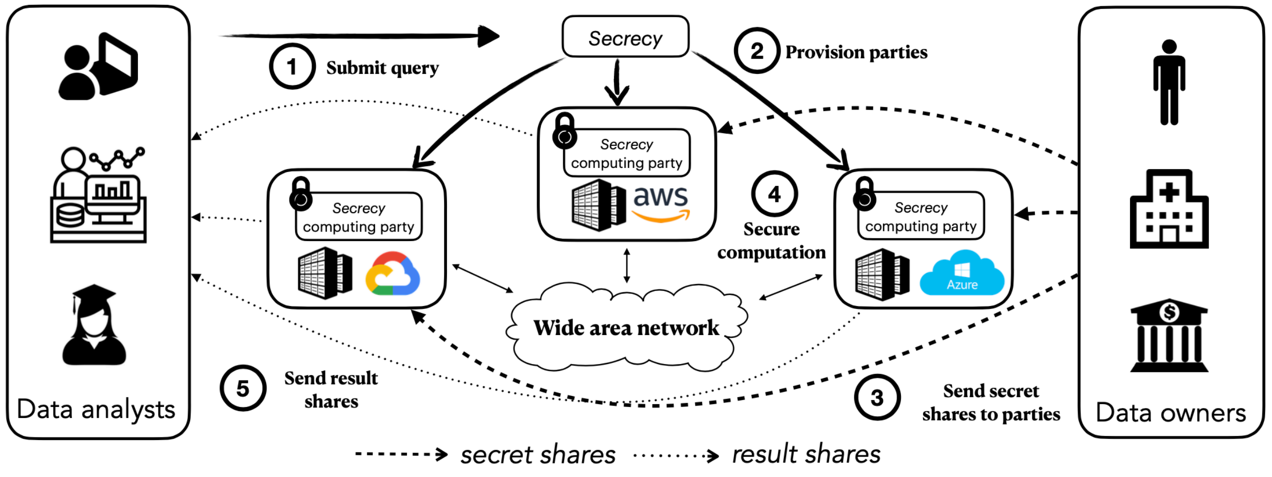


Figure 1: Overview of secure collaborative analytics with SECRECY. Clients (e.g., data analysts) access a catalog with metadata about available private datasets and submit public queries to the SECRECY service. SECRECY performs query planning and provisions computing parties in available non-colluding cloud providers, e.g., GCP, AWS, and Azure. Next, it instructs data owners to distribute *secret shares* (cf. §3) of their data to computing parties. Parties execute the query under MPC and send the results to the analysts. SECRECY considers adversaries who may have complete control over the network. All data remain private as long as an adversary does not compromise the majority of cloud providers.

and automated query planning. However, to achieve practical performance, these works employ optimizations that either leak information to untrusted parties or apply to peer-to-peer settings where data holders also serve as computing parties using trusted resources (we provide more details in §8). Outsourced MPC, on the other hand, removes the computation burden from data holders and has recently gained attention, especially in industry, with systems like Prio [45] Carbyne [11], CrypTen [70] and Cape Privacy's TF Encrypted [1]. Yet, these frameworks focus on certain statistics or ML workloads and do not support general-purpose analytics.

To fill this gap, we present SECRECY, a new relational MPC system for efficient collaborative analytics in the cloud with no information leakage. In SECRECY we take a fundamentally different approach over prior work and we carefully co-design the MPC protocol, query engine, and distributed runtime into a single platform. SECRECY's core novelty is a generic cost-based optimization framework for relational MPC that does not rely on trusted infrastructure. As such, it enables data holders and analysts to use untrusted cloud resources on demand and benefit from the "pay-as-you-go" model while retaining the *full* security guarantees of the cryptographic protocols.

Contributions. We make the following contributions:

- We present a relational MPC system, based on secret sharing, that enables efficient collaborative analytics with no information leakage.
- We design vectorized MPC primitives and relational operators that amortize the network I/O of secret sharing. Contrary to prevailing wisdom, we show that this approach can be competitive with widely adopted MPC techniques for relational analytics in both LAN and WAN environments.
- We define an analytical cost model that formulates MPC query costs in terms of secure computation and communication operations. We use this cost model to develop a novel query optimization framework for relational MPC.
- We implement a Volcano-style query processor that leverages the cost model to automatically apply a rich set of logical, physical, and protocol-aware optimizations which can improve performance by orders of magnitude.
- We evaluate SECRECY's performance and the effectiveness of its optimizations using real and synthetic queries. Our experiments show that SECRECY outperforms state-of-the-art MPC frameworks and scales to much larger datasets.

We believe SECRECY will become a valuable tool to cloud providers, data holders, and analysts by enabling new privacy-preserving applications and marketplaces on existing cloud infrastructure. We will release SECRECY as open-source [9].

2 SECRECY system overview

Figure 1 presents an overview of the SECRECY cloud service. Each party in SECRECY has one or more of the following roles: (i) *data holder or data owner* who provides some input data, (ii) *computing party*, e.g., a cloud provider that provides resources to perform the secure computation, and (iii) *analyst* who issues a query to learn the result. SECRECY supports any number of data owners and uses three computing parties. A "party" is a logical entity and does not necessarily correspond to a single compute node. SECRECY does not make any assumption about the physical deployment: parties can be deployed in private clusters, in a multi-cloud, or across multiple providers in a hybrid or federated cloud.

2.1 Design principles

We have designed SECRECY on the following principles:

1. No information leakage. SECRECY reveals nothing about the input, output, or intermediate data and the execution metadata to untrusted parties, including the cloud providers. It completely hides access patterns, intermediate, and output result sizes. SECRECY does not require data owners to annotate attributes as sensitive or non-sensitive and does not try to sidestep the secure computation. It executes all operations under MPC and protects all attributes to prevent inference attacks that exploit correlations or functional dependencies in the data which may be unknown to data owners.

2. No reliance on trusted execution environments. SECRECY does not rely on any (semi-)trusted party, honest broker or specialized secure hardware. To remove barriers for adoption, we target general-purpose compute and cloud.

3. Decoupling of roles. In SECRECY, a party may have any combination of roles, that is, data owners can (but do not have to) also act as computing parties and/or analysts without affecting the security guarantees. Query optimization in SECRECY does not rely on data ownership and does not require data owners to participate in MPC using trusted resources. Due to decoupling, SECRECY can effectively use a small number of computing parties to support any number of data owners without affecting the scalability of secure computation.

4. High expressivity. SECRECY's protocol does not pose any restriction on the types of queries that can be supported. While there exist many efficient protocols for specific instances of MPC operators, these are often not composable. In SECRECY, we have decided to provide general operator implementations that are independent of the data characteristics and can be composed with each other to create arbitrary query plans.

2.2 Threat model and security guarantees

SECRECY protects data throughout the entire lifecycle and treats the query itself as *public*, i.e., it assumes that data owners and analysts have previously agreed on a relational query to compute and this query is known to the computing parties, as in prior works [22–24, 95, 105]. To evaluate the query, SECRECY servers execute an identical computation and exchange messages with each other according to a protocol. All communication in a SECRECY deployment must be done via authenticated and encrypted channels (e.g., using TLS).

Threat model. SECRECY assumes "honest-but-curious" parties and can withstand adversaries who have two types of capabilities. First, adversaries have complete control over the network and can monitor all network links. Second, adversaries may compromise one computing party and can see all of its internal state (e.g., memory contents, access patterns, and data sent/received) but without altering its execution. That said, most of the techniques we present in this work are also compatible with malicious-secure MPC protocols where parties can deviate from the protocol arbitrarily (cf. §5.4).

Security guarantees. We have purposely designed SECRECY in a modular fashion to ensure it can directly inherit all security guarantees of the underlying MPC protocol. SECRECY relies on the semi-honest 3-party replicated secret sharing protocol by Araki et al. [17, 81]. The protocol provides two types of guarantees: (i) *privacy*, meaning that computing parties do not learn anything about the data, and (ii) *correctness*, meaning that all participants are convinced that the computation output is accurate. As long as the computing parties do not collude, the SECRECY *servers cannot learn anything beyond the size of the input data* (which can also be padded by the data owners). Only the designated analysts learn the result of the query. SECRECY does not use differential privacy to protect the result from possible inference attacks by the analysts but it could be easily augmented to do so (cf. §8). We also stress that formal verification of the SECRECY software is out of scope for this work (but an exciting future direction). For a detailed security analysis please refer to Appendix B.

2.3 Cost-based secure query optimization

Cost-based query optimization on plaintext data relies on selectivity estimations to reduce the size of intermediate results. MPC operators, however, are *oblivious*, i.e., their control flow is independent of the input data and incurs exactly the same accesses for all inputs of the same size. Oblivious operators do not reveal the size of intermediate data to prevent reconstruction attacks based on selectivity statistics [60, 69, 84]. As a consequence, traditional selectivity-based techniques for plaintext queries [41], such as join reordering or filter pushdown, are not effective when optimizing plans under MPC. For instance, given that oblivious selections do not reduce the size of intermediate data, pushing a filter down does not improve the cost of subsequent operators in the plan.

To devise effective optimizations under MPC, we express the plan costs in terms of secure computation and communication operations. In SECRECY, we define three types of costs:

- The **operation cost**, C_o, which is determined by the number of primitive MPC operations per party. Primitive operations can be *local* ($+, \oplus$), which do not require communication, or *remote* ($\times$, $\wedge$), which require some message exchange between parties (cf. §3).
- The **synchronization cost**, C_s, given by the number of communication rounds across parties that are inherent in MPC. Each round corresponds to a barrier, i.e. a synchronization point in the distributed execution, where parties must exchange data in order to proceed.
- The **cost of composition**, C_c, which is also measured in operations and communication rounds required to compose oblivious relational operators under MPC.

SECRECY applies automatic optimizations that aim to minimize at least one of these three costs. We present a comprehensive cost analysis of oblivious operators and their composition in §4. Contrary to plaintext query optimization where estimations are often erroneous [76], in MPC we can use the typical dynamic programming approach from database optimizers [98] to compute *exact* plan costs at compile time, since C_o, C_s, and C_c do not depend on the data distribution.

3 Background on MPC

MPC protocols follow one of two general techniques: obscuring the truth table of each operation using Yao's *garbled circuits* [114], or performing operations over encoded data using *secret sharing* [100]. So far, garbled circuits has been the preferred method to securely compute Boolean circuits in high-latency environments, as they only need a small (constant) number of rounds between computing parties at the cost of incurring a large memory overhead [73]. On the other hand, secret sharing-based approaches require more rounds (that depend on the input size) but have a small memory footprint and consume less overall bandwidth. In this work we employ secret sharing in the honest-majority setting that is reasonable for many real use cases [20,25,26,28,101]. Looking ahead, in §7 we will demonstrate that SECRECY's optimizations make secret sharing competitive in both LAN and WAN settings.

3.1 Replicated secret sharing

SECRECY encodes an ℓ-bit string of sensitive data s by splitting it into three *secret shares* s_1, s_2, and s_3 that individually have the uniform distribution over all possible ℓ-bit strings (for privacy) and collectively suffice to specify s (for correctness). Computing parties are placed on a *logical ring* and each party P_i receives two of the shares s_i and s_{i+1} (i.e., P_1 receives s_1, s_2, P_2 receives s_2, s_3, and P_3 receives s_3, s_1). Hence, any two parties can reconstruct a secret if they collude, but any single party cannot, no matter how powerful it is. SECRECY supports two secret sharing formats (and can also transition from one to the other): *boolean* secret sharing in which $s = s_1 \oplus s_2 \oplus s_3$, where $\oplus$ denotes the bitwise XOR operation, and *additive* or *arithmetic* secret sharing in which $s = s_1 + s_2 + s_3 \bmod 2^\ell$.

3.2 Oblivious primitives

In this section, we provide an overview of the oblivious primitives we use throughout our work. Let s, t be two secrets, and $op(s,t)$ an operation on these secrets. The primitives allow SECRECY servers to start with shares of s and t and jointly compute shares of the result $op(s,t)$ without learning anything about s and t. Each server can use these shares in subsequent operations or send them to the analysts, who can reconstruct the true output of $op(s,t)$. We stress that our oblivious relational operators in §4 are agnostic of the underlying primitives and it would be perfectly possible to implement primitives based on other MPC protocols without affecting the applicability of the optimizations in §5.

Basic operations. When given boolean secret-shared data corresponding to ℓ-bit strings s and t, parties can compute shares of the bitwise XOR $s \oplus t$ locally (i.e., without communication) and shares of the bitwise AND st with synchronization cost equal to $C_s(\texttt{AND}) = 1$ round of communication. The operation cost is the same in both cases, i.e., $C_o(\texttt{XOR}) = C_o(\texttt{AND}) = \ell$ (we consider 1-bit boolean operations to have unit cost). Similarly, given two secrets u and v that have been arithmetically shared, parties can compute shares of the sum $u+v$ locally (similarly to XOR) and shares of the product $u \cdot v$ with 1 communication round (similarly to the bitwise AND operation).

Mixed-mode operations. The above boolean and arithmetic operations are universal and can be used to compute *any* function. Moreover, there exist well-known constructions of several specific operations with fast instantiations based on boolean and/or arithmetic sharing. In SECRECY we implement several such oblivious operations: (in)equality, a compare-and-swap multiplexer, boolean addition with a ripple-carry adder, boolean-to-arithmetic conversion, and more.

For details on the SECRECY primitives, please refer to §A.1.

4 SECRECY operators and cost model

In this section, we first provide an overview of oblivious operators in SECRECY along with their asymptotic costs (§4.1) and the costs of their composition under MPC (§4.2). We then explain how SECRECY computes exact plan costs in §4.3. Although, in practice, the SECRECY planner uses the detailed cost formulas from Appendix A, knowledge of the asymptotic costs is sufficient to follow the optimizations in §5.

4.1 Oblivious relational operators

SECRECY supports a rich class of oblivious relational operators: `SELECT`, `PROJECT`, `(SEMI-) JOIN`, `GROUP-BY`, `DISTINCT`, and `ORDER-BY` with `LIMIT`. It also supports the following aggregations under MPC: `COUNT`, `SUM`, `MIN/MAX`, and global `AVG`.

All operators have the same semantics as their plaintext counterparts but their control flow is data independent; in practice, this means that the SECRECY code does not have any `if` statements that depend (either directly or indirectly) on the input data. At a high level, oblivious selection requires a linear scan over the input relation, join and semi-join operators require a nested-loop over the two inputs, whereas order-by, distinct, and group-by are based on a sorting network.

In all cases, operator predicates can be arbitrary logical expressions with atoms that may also include arithmetic expressions $(+, \times, =, >, <, \neq, \geq, \leq)$ and are evaluated under MPC

Operator	#operations (#messages)	#communication rounds
SELECT	$O(n)$	$O(1)$
JOIN	$O(n \cdot m)$	$O(1)$
SEMI-JOIN	$O(n \cdot m)$	$O(\log m)$
ORDER-BY	$O(n \cdot \log^2 n)$	$O(\log^2 n)$
DISTINCT	$O(n \cdot \log^2 n)$	$O(\log^2 n)$
GROUP-BY	$O(n \cdot \log^2 n)$	$O(\log^2 n)$
MASK	$O(n)$	$O(1)$

Table 1: Summary of operation (C_o) and synchronization costs (C_s) for general oblivious relational operators *w.r.t.* the cardinalities (n, m) of the input relation(s). The asymptotic number of operations equals the asymptotic number of messages per computing party, as each individual operation on secret shares involves a constant number of message exchanges under MPC. Independent messages can be batched in rounds as shown in the rightmost column.

using the oblivious primitives of §3.2. All operators except `PROJECT` and `ORDER-BY` append a new attribute to each record of their input relation that stores a (secret-shared) *valid bit*: this bit denotes whether the record belongs to the output of the operator and is computed under MPC.

Table 1 shows the asymptotic operation and synchronization costs per operator with respect to the input size. `MASK` is a special operator used by SECRECY to hide records (with "garbage" values) upon a condition. The formal operator semantics and their exact costs are given in §A.2.

4.2 Composing oblivious operators

We define the composition of two operators as applying the second operator to the output of the first. One merit of our approach is that all operators of §4.1 reveal nothing about their output or access patterns and can be arbitrarily composed into an *end-to-end oblivious* plan without special treatment.

Let op_1 and op_2 be two SECRECY operators. In general, the composition $op_2(op_1(R))$ has an *extra cost* (additional to the cost of applying the operators op_1 and op_2) as it requires evaluating under MPC a logical expression e_c for each generated tuple. We define the *composition cost* of $op_2(op_1(R))$ as the cost of evaluating e_c on all records generated by op_2. The expression e_c depends on the types of operators. For example, composing two selections, each one appending a valid bit to the input relation, requires ANDing the two bits for each record. Table 2 shows the asymptotic composition costs for different operator pairs. The detailed costs are given in §A.3.

Note that applying the distinct operator to the output of a selection, a group-by or a (semi-)join requires a linear number of rounds. This is a significant increase over the $O(\log^2 n)$ rounds required by distinct when applied to a base relation (cf. Table 1). In §5.2, we propose an optimization that reduces the cost of these compositions to a logarithmic factor.

4.3 Computing optimal plan costs

SECRECY's query planner is based on a typical bottom-up dynamic programming algorithm [98] that computes optimal

Operator pair(s)	#rounds
{SELECT, (SEMI-)JOIN, GROUP-BY, DISTINCT} $\longrightarrow$ DISTINCT	$O(n)$
DISTINCT $\longrightarrow$ {SELECT, (SEMI-)JOIN}	$O(1)$
SELECT $\leftrightarrow$ (SEMI-)JOIN	$O(1)$
GROUP-BY $\longrightarrow$ {SELECT, (SEMI-)JOIN}	$O(1)$
{SELECT, (SEMI-)JOIN, DISTINCT, GROUP-BY} $\longrightarrow$ GROUP-BY	$O(\log^2 n)$

Table 2: Summary of composition costs (C_c) in number of rounds for pairs of operators in SECRECY *w.r.t* the number of generated records (n). Arrows denote the order of applying the two operators. Composition incurs a small constant number of boolean operations per record, so its cost in number of operations is $O(n)$ in all cases.

plans based on our analytical cost model and a set of transformation rules that we present in §5. The algorithm identifies all operators in the input query and proceeds in stages: at each stage it creates bigger plans by adding a new operator to sub-plans from the previous stage. Initially, the set of possible sub-plans includes scans of the input relations. When creating a new (sub-)plan, the algorithm checks for all applicable transformation rules and applies them exhaustively to generate equivalent (sub-)plans with lower cost. The cost of a plan is computed as follows. Each time an operator op is added to a sub-plan, SECRECY computes the operation and synchronization costs $C_o(op)$ and $C_s(op)$. If the operator is applied to the output of another operator, SECRECY also computes the composition cost C_c. To do so, it augments the current plan with a special operator $op_{e_c}(op_i(op_j(..)))$ that applies the composition predicate e_c (§4.2). $C_o(op_{e_c})$ and $C_s(op_{e_c})$ amount to the cost of composing the operators op_i and op_j in number of operations and rounds respectively. For a plan with k operators, the total cost is $\sum_{i=1}^{k} \alpha C_o(i) + \beta C_s(i)$, where α, β are parameters of the deployment. The algorithm returns the plan with the minimum cost from the final stage.

5 SECRECY optimizations for relational MPC

Here we present the optimizations we introduce in SECRECY to speed up MPC query execution: (i) logical transformation rules, such as operator reordering and decomposition (§5.1), (ii) physical optimizations, such as operator fusion, vectorized primitives and message batching (§5.2), and (iii) secret-sharing optimizations that further reduce the number of communication rounds for certain operators (§5.3). Table 3 summarizes the notation used in the remainder of the paper.

Target queries. Our work focuses on collaborative analytics under MPC where two or more data owners want to outsource queries on their collective data without compromising privacy. We consider all inputs as sensitive and assume that data owners wish to protect their raw data and avoid revealing attributes of base relations in query results. For example, employing MPC to compute a query that includes patient names along with their diagnoses in the `SELECT` clause is pointless. Thus, we target queries that return global or per-group aggregates and/or distinct results, as in prior works.

Symbol	Description
ℓ	length of the share representation in bits
R, S	relations with cardinality $\vert R\vert$ and $\vert S\vert$
$\sigma_\phi(R)$	selection with predicate ϕ
$R \bowtie_\theta S$	join with predicate θ
$R \ltimes_\theta S$	left semi-join with predicate θ
$\delta_a(R)$	distinct operator on attribute a
$\gamma_a^g(R)$	group-by operator on attribute a with aggregation function g
$s_{\uparrow a}(R)$	sort on attribute a (ascending)

Table 3: Notation used in the paper

5.1 Logical transformation rules

SECRECY uses three types of logical transformations that reorder and decompose operators to reduce the MPC costs:

5.1.1 Blocking operator push-down

Blocking oblivious operators (GROUP-BY, DISTINCT, ORDER-BY) materialize and sort their entire input before producing any output tuple. Contrary to a plaintext optimizer that would most likely place sorting after selective operators, in MPC we have an incentive to push blocking operators down, as close to the input as possible. Since oblivious operators do not reduce the size of intermediate data, sorting the input is clearly the best option. Blocking operator push-down can provide considerable performance improvements in practice, even if the asymptotic costs do not change. As an example, consider the rule that pushes ORDER-BY before a selection, i.e., $s_{\uparrow a}(\sigma_\phi(R)) \rightarrow \sigma_\phi(s_{\uparrow a}(R))$. Although this rule would not generate a more efficient plan in plaintext evaluation, it does so in the MPC setting. This is because the operations required by the oblivious ORDER-BY depend on the cardinality and the number of attributes of the input relation. Applying the selection after the order-by reduces the actual (but not the asymptotic) operation cost, as σ_ϕ appends one attribute to R.

Applicability. Rules in this class are valid relational transformations with no special applicability conditions under MPC.

5.1.2 Join push-up

The second class of rules leverage the fact that JOIN is the only operator whose output is larger than its input. Based on this, we have an incentive to perform joins as late as possible in the query plan so that we avoid applying other operators to join results, especially those that require materializing the join output. For example, placing a blocking operator after a join requires sorting the cartesian product of the input relations, which increases the operation cost of the blocking operator to $O(n^2 \log^2 n)$ and the synchronization cost by 4×.

Example. Consider the following query:

```
Q1: SELECT DISTINCT R.id
    FROM R, S
    WHERE R.id = S.id
```

and the rule $\delta_{id}(R \bowtie_{id=id} S) \rightarrow \delta_{id}(R) \bowtie_{id=id} \delta_{id}(S)$. Let R and S have the same cardinality n. A plan that applies

```
1  $s_{\uparrow a_\theta \uparrow a_k}(R)$;      //sort input relation $R$ on $a_\theta$, $a_k$
2  let $d \leftarrow |R|/2$;   //Distance of tuples to aggregate
3  while $d \geq 1$ do
4      for each pair of tuples $(t_i, t_{i+d})$, $0 \leq i < |R| - d$, do
           //Are tuples in the same group?
5          let $b \leftarrow t_i[a_k] \overset{?}{=} t_{i+d}[a_k]$;
           //Are tuples in semi-join output too?
6          let $b_c \leftarrow b \wedge t_i[a_\theta] \wedge t_{i+d}[a_\theta]$; //$b_c$ is a bit
           //Oblivious aggregation via multiplexing
7          $t_i[a_g] \leftarrow b_c \cdot \left(t_i[a_g] + t_{i+d}[a_g]\right) + (1 - b_c) \cdot t_i[a_g]$;
8          $t_{i+d}[a_v] \leftarrow \neg b_c$;      //$a_v$ is the valid bit
9          mask $t_{i+d}$ when $t_{i+d}[a_v] = 0$;
10     $d = d/2$;
11 mask remaining tuples with $t[a_v] = 0$ and shuffle $R$;
```

Algorithm 1: 2nd phase of Join-Aggregation decomposition

DISTINCT after the join operator requires $O(n^2 \log^2 n)$ operations. On the other hand, pushing DISTINCT before JOIN reduces the operation cost to $O(n^2)$ and the composition cost from $O(n^2)$ to $O(1)$ in number of rounds. The asymptotic synchronization cost is the same for both plans, i.e. $O(\log^2 n)$, but the actual number of rounds when DISTINCT is pushed before JOIN is 4× lower.

Applicability. Rules in this class have the same applicability conditions as similar rules for plaintext queries [42, 113], even though their goal is different. In our setting, the re-orderings do not aim to reduce the size of intermediate data. In fact, a plan that applies DISTINCT on a JOIN input produces exactly the same amount of intermediate data as a plan where DISTINCT is placed after JOIN, yet our analysis reveals that the second plan has higher MPC costs.

5.1.3 Join-Aggregation decomposition

Consider a query plan where a JOIN on attribute a_j is followed by a GROUP-BY on another attribute $a_k \neq a_j$. In this case, pushing the GROUP-BY down does not yield a semantically equivalent plan. Still, we can optimize the plan by decomposing the aggregation in two phases and push the first and most expensive phase of GROUP-BY before the JOIN.

Let R, S be the join inputs, where R includes the group-by key a_k. The first phase of the decomposition sorts R on a_k and computes a semi-join on a_j that appends two attributes to R: the valid bit a_θ introduced by the semi-join, and a second attribute a_g that stores the result of a partial aggregation[1] (we come back to this later).

In the second phase, we compute the final aggregates per a_k using Algorithm 1, which takes into account the attribute a_θ and updates the partial aggregates a_g in-place using odd-even aggregation. The decomposition essentially replaces the join

[1]In case the aggregation function is AVG, we need to keep the value sum (numerator) and count (denominator) as separate secret-shared attributes in R.

with a semi-join and a partial aggregation in order to avoid performing the aggregation on the cartesian product $R \times S$. This way, we significantly reduce the number of operations and communication rounds, but also ensure that the space requirements remain bounded by $|R|$, since the join output is not materialized. Note that this optimization is fundamentally different than performing a partial aggregation in the clear (by the data owners) and then computing the global aggregates under MPC [22, 95]; in our case, all data are secret-shared amongst parties and *both* phases are under MPC.

Example. Consider the following query:

```
Q2: SELECT R.a_k, COUNT(*)
    FROM R, S
    WHERE R.id = S.id
    GROUP BY R.a_k
```

Let R and S have the same cardinality n. The plan that applies GROUP-BY to the join output requires $O(n^2 \log^2 n)$ operations and $O(\log^2 n)$ communication rounds. When decomposing the aggregation $\gamma_{a_k}^{\texttt{COUNT(*)}}$ in two phases, the operation cost is reduced to $O(n^2)$ and the synchronization cost is 4× lower. The space requirements are also reduced from $O(n^2)$ to $O(n)$. In our example, the partial aggregation corresponds to the function $t[a_\theta] = \sum_{\forall t' \in S} \theta(t,t'), t \in R$, where $\theta(t,t') := t[\texttt{id}] \overset{?}{=} t'[\texttt{id}]$. Similar partial aggregations can be defined for SUM, MIN/MAX, and AVG.

Decomposition with DISTINCT. A similar idea can also be employed when the join is followed by a DISTINCT. The transformation rule in this case is $\delta_{R.a}(R \bowtie_\theta S) \rightarrow \delta'_a(s_{\uparrow R.a_\theta, \uparrow R.a}(R \ltimes_\theta S))$, where a_θ denotes the semi-join bit and $\delta'(\cdot)$ is the final phase of distinct that compares adjacent tuples with $a_\theta = 1$. For example, the plan $\delta_{R.a}(R \bowtie_{b=b} S)$ can be replaced with the equivalent plan $\delta'_{R.a}(s_{\uparrow R.a_\theta, \uparrow R.a}(R \ltimes_{b=b} S))$ to reduce the operation cost from $O(n^2 \log^2 n)$ to $O(n^2)$ and the synchronization cost from $O(n^2)$ to $O(\log^2 n)$.

Applicability. The decomposition technique we described is applicable to any θ-join followed by (i) a GROUP-BY with aggregation or (ii) a DISTINCT operator, under the condition that the group-by or distinct keys belong to one join input.

5.2 Physical optimizations

We now describe a set of physical optimizations in SECRECY.

5.2.1 Predicate fusion

Fusion is a common optimization in plaintext query planning, e.g., when predicates of multiple filters are merged and executed by a single operator. Fusion has been recently used to speed up secure ML pipelines in Cerebro [117] and is also applicable to oblivious relational operators. In our setting, fusion is achieved by identifying independent operations that can be executed efficiently within the same communication round. For example, if the equality check of an equi-join and a selection are independent of each other, a fused operator requires $\lceil \log \ell \rceil + 1$ rounds instead of $2\lceil \log \ell \rceil + 1$ (cf. §A). Next, we describe a somewhat more interesting case of fusion.

5.2.2 Operator fusion

Recall that applying DISTINCT after SELECT requires n communication rounds (§4.2). We can avoid this overhead by fusing the two operators in a different way, that is, sorting the input relation on the selection bit first and then on the distinct attribute. Sorting on two (instead of one) attributes adds a small constant factor to each oblivious compare-and-swap operation, hence, the asymptotic complexity of the sorting step remains the same. When distinct is applied to the output of other operators, including selections and (semi-)joins, this physical optimization keeps the number of rounds required for the composition low.

Example. Consider the following query:

```
Q3: SELECT DISTINCT id
    FROM R
    WHERE a_k = 'c'
```

Fusing the distinct and selection operators reduces the number of communication rounds from $O(n)$ to $O(\log^2 n)$, as if the distinct operator was applied only to R (without a selection). DISTINCT can be fused with a join or a semi-join operator in a similar way. In this case, the distinct operator takes into account the (semi-)join bit.

5.2.3 Vectorization and message batching

In secret sharing protocols, non-local operations require exchanging very small messages. Applying multiple such independent operations in a vectorized fashion and exchanging the respective messages in bulk improves performance tremendously. Consider applying a selection with an equality predicate on a relation with n tuples. Performing oblivious equality on one tuple requires $\lceil \log \ell \rceil$ rounds. Applying the selection tuple-by-tuple and sending messages eagerly (as soon as they are generated) results in $n \cdot \lceil \log \ell \rceil$ rounds. Instead, if we apply independent selections across the entire relation and exchange messages in bulk, we can reduce the total synchronization cost to $\lceil \log \ell \rceil$. We have designed all SECRECY primitives to apply vectorization and message batching by default, otherwise the cost of secret sharing is prohibitive. Costs in Tables 1 & 2 already take message batching into account.

5.3 Secret-sharing optimizations

Here we propose optimizations that take advantage of mixed-mode MPC protocols which permit both arithmetic and boolean computations. While SECRECY uses boolean secret sharing for most operations, computing arithmetic expressions or aggregations like COUNT and SUM on boolean shares requires using a ripple-carry adder (RCA), which in turn requires many communication rounds. Performing these operations on additive shares would require no communication, but converting shares from one format to another can be expensive. Below, we describe two optimizations that avoid the RCA in aggregations and predicates with constants.

5.3.1 Dual sharing

The straight-forward approach of switching from boolean to additive shares (and vice versa) based on the type of operation does not pay off; the conversion itself relies on RCA, which has to be applied twice to switch to the other representation and back. The cost-effective way would be to evaluate logical expressions using boolean shares and arithmetic expressions using additive shares. However, this is not always possible because arithmetic and boolean expressions in oblivious queries often need to be composed into the same formula. We mitigate this problem using a dual secret-sharing scheme.

Recall the example query **Q2** from §5.1.3 that applies an aggregation function to the output of a join according to Algorithm 1. The attribute a_θ in Algorithm 1 is a single-bit attribute denoting that the respective tuple is included in the join result. During oblivious evaluation, each party has a boolean share of this bit that is used to compute the arithmetic expression in line **6**. The naïve approach is to evaluate the following equivalent logical expression directly on the boolean shares of b_c, $t_i[a_g]$, and $t_{i+d}[a_g]$:

$$t_i[a_g] \leftarrow b_\ell \wedge \mathtt{RCA}\Big(t_i[a_g], t_{i+d}[a_g]\Big) \;\oplus\; \overline{b_\ell} \wedge t_i[a_g]$$

where RCA is the oblivious ripple-carry adder primitive, b_ℓ is a string of ℓ bits (the length of a_g) all of which are set equal to b_c, and $\overline{b_\ell}$ is the binary complement of b_ℓ. Evaluating the above expression requires ℓ communication rounds for RCA plus two more rounds for the logical ANDs ($\wedge$). On the contrary, SECRECY evaluates the equivalent formula in line **6** of Algorithm 1 in four rounds (independent from ℓ) as follows. First, parties use arithmetic shares for the attribute a_g to compute the addition locally. Second, each time they compute the bit b_c in line **5**, they exchange boolean as well as arithmetic shares of its value. To do this efficiently, we rely on the single-bit conversion protocol also used in CrypTen [70], which requires two rounds of communication. Having boolean and arithmetic shares of b_c allows SECRECY to use it in boolean and arithmetic expressions without paying the cost of RCA.

5.3.2 Proactive sharing

The previous optimization relies on b_c being a single bit. In many cases, however, we need to compose boolean and additive shares of arbitrary values. Representative examples are join predicates with arithmetic expressions on boolean shares, e.g. $(R.a - S.a \geq c)$, where a is an attribute and c is a constant. We can speedup the oblivious evaluation of such predicates by proactively asking the data owners to send shares of the expression results. In the previous example, if parties receive boolean shares of $S.a + c$ they can avoid computing the boolean addition with RCA. A similar technique is also applicable for selection predicates with constants. In this case, to compute $a > c$, if parties receive shares of $a - c$ and $c - a$, they can transform the binary equality to a local comparison with zero. Note that proactive sharing is fundamentally different than having data owners perform local filters or pre-aggregations prior to sharing. In the latter case, the computing parties might learn the selectivity of a filter or the number of groups in an aggregation (if results are not padded). In our case, parties simply receive additional shares and will not learn anything about the intermediate query results.

5.4 Generality of optimizations

The logical and physical query optimizations constructed in this work (§5.1-5.2) apply generally to any mixed-mode MPC protocol that supports the primitives we describe in §3.2. This includes protocols that remain secure in the face of a malicious adversary who can deviate from the protocol arbitrarily (e.g., [46, 71, 89]), and (authenticated) garbled circuit protocols [109, 114] combined with conversions to arithmetic secret sharing [50, 89] as needed. SECRECY can also support alternative instantiations of oblivious primitives with different cost profiles, such as constant-round equality and comparisons with higher operation costs [48, 86].

While the secret-sharing optimizations of §5.3 are specific to SECRECY's underlying MPC protocol (§3.1), we expect that similar techniques can be developed also for other protocols. Extending the SECRECY planner to consider the cost profiles of various building blocks is an exciting avenue for future work. We provide a formal discussion of generality in Appendix B.

6 SECRECY implementation

Despite a rich open-source ecosystem of general-purpose MPC frameworks [62], we found that existing tools either lack support for general relational operations (with θ-predicates) or cannot effectively amortize network I/O. For these reasons, we implemented SECRECY in C/C++, entirely from scratch. We designed our secure primitives to operate directly on relations and we also built a library of general oblivious relational operators that can be combined into arbitrary query plans.

System overview. Figure 2 shows the SECRECY architecture and software stack. Data analysts submit queries through a client application that exposes a SQL interface and provides a query planner that performs query rewriting and cost-based optimization. Data owners use the secret-sharing generation module to distribute random shares of their data to the computing parties. Computing parties can be deployed on premises,

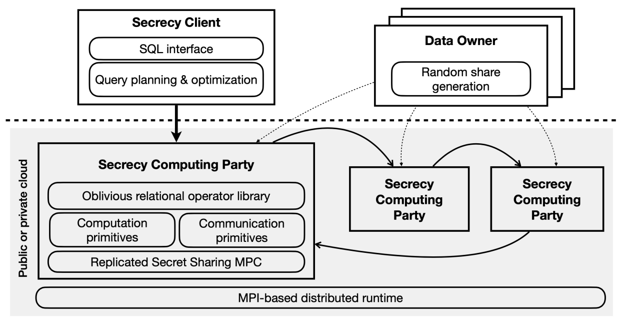


Figure 2: The SECRECY system consists of (i) a client application that can be used by data analysts to submit queries, (ii) a data owner application to generate and distribute secret shares, and (iii) three computing parties that execute queries under MPC.

in a hybrid cloud, or across multiple clouds. To automate cloud deployment we use Ansible [7]. The parties' software stack consists of (i) a custom implementation of the replicated secret sharing protocol, (ii) a library of secure computation and communication primitives, and (iii) a library of oblivious relational operators. The distributed runtime and communication layer are based on MPI [5]. Each party is a separate MPI process that handles both computation and communication.

Operator pipelining. SECRECY relational operators and secure primitives are designed to process table rows in batches. The batch size is configurable and allows SECRECY to compute expensive operators, such as joins, with full control over memory requirements. While batching does not reduce the total number of operations, we leverage it to compute on large inputs in a pipelined fashion, without running out of memory or switching to a disk-based evaluation.

Query planning and execution. Upon startup, the parties establish connections to each other and learn the process IDs of other parties. Next, they receive shares for each input relation from the data owners. Queries are specified either in SQL (and go through query planning) or in a declarative DSL that allows seamless operator composition by abstracting MPC details. For SQL parsing we use the Hyrise parser [12]. SECRECY's planner generates the optimal query plan as explained in §4.3. To evaluate a query, parties execute the same oblivious physical plan on their random shares and return the results to the designated client. We use a 64-bit share representation by default, so $\ell = 64$ (cf. Table 3).

7 Experimental evaluation

Our experimental evaluation is structured into four parts:

Benefits of query optimization. In §7.2, we evaluate the benefits of SECRECY's optimizations on eight real and synthetic queries. We show that SECRECY's cost-based optimizer reduces the runtime of complex queries by up to three orders of magnitude both in a LAN and a multi-cloud setting.

Performance on real and synthetic queries. In §7.3 we evaluate SECRECY's performance as input sizes grow. We use queries that include selections, group-by, distinct, semi-join, and theta-joins with both equality and inequality predicates. Our results demonstrate that SECRECY can scale to millions of input rows and evaluate complex queries in reasonable time with modest use of resources.

Micro-benchmarks. In §7.4, we evaluate individual logical, physical, and secret-sharing optimizations on the three queries from §5.1-5.3. Our results demonstrate that pushing down blocking operators reduces execution time by up to 1000× and enables queries to scale to 100× larger inputs. Further, we show that operator fusion and dual sharing improve execution time by an order of magnitude in the WAN setting.

Comparison with state-of-the-art frameworks. In §7.5, we compare SECRECY with SMCQL [22] and the 2-party semi-honest version of EMP [108]. We choose SMCQL (the ORAM-based version) as the only open-source relational framework with semi-honest security and no leakage. We also choose the EMP library since it is used by all recent systems, namely Shrinkwrap [23], SAQE [24], a new version of SMCQL, and Senate [95]. Although none of these systems is publicly available, they all build their relational MPC engines on top of EMP. We show that SECRECY outperforms them both and can comfortably process much larger datasets.

We provide additional micro-benchmarks and experiments with EMP in Appendix D.

7.1 Evaluation setup

We use three cloud deployments: (i) `AWS-LAN` uses an *EC2 r5.xlarge* instance per party in the `us-east-2` region, (ii) `AWS-WAN` distributes parties across `us-east-2` (Ohio), `us-east-1` (Virginia), and `us-west-1` (California), and (iii) `MULTI-CLOUD` distributes parties across three different cloud providers, namely AWS (Ohio), Google Cloud (South Carolina), and Azure (Virginia). VMs have 32GB of memory and run Ubuntu `20.04`, `C99`, `gcc 5.4.0`, and `MPICH 3.3.2`. Measurements are averaged over at least three runs and plotted in log-scale, unless otherwise specified.

Queries. We use 11 queries for evaluation, including five real-world queries from previous MPC works [22–24, 95, 105]. Three are medical queries [22]: *Comorbidity* returns the ten most common diagnoses of individuals in a cohort, *Recurrent C.Diff.* returns the distinct ids of patients who have been diagnosed with `cdiff` and have two consecutive infections between 15 and 56 days apart, and *Aspirin Count* returns the number of patients who have been diagnosed with heart disease and have been prescribed aspirin after the diagnosis was made. We also use queries from other MPC application areas [95]: *Password Reuse* asks for users with the same password across different websites, while *Credit Score* asks for persons whose credit scores across different agencies have

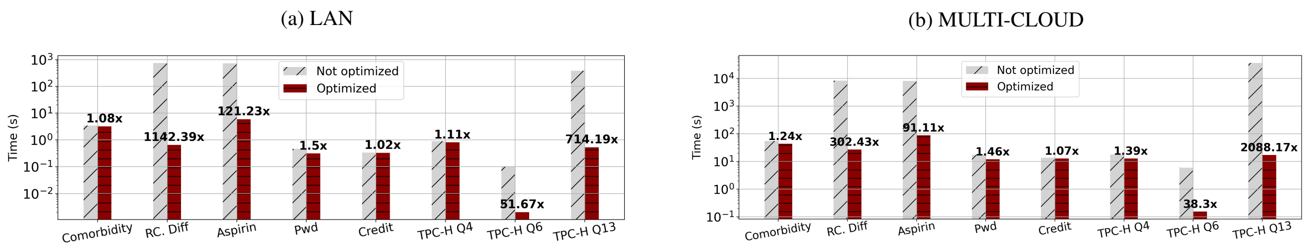


Figure 3: SECRECY end-to-end performance when optimizations are enabled (`Optimized`) and disabled (`Not optimized`) for real and synthetic queries. Logical and physical optimizations result in over 1000× lower execution times, while secret-sharing optimizations improve performance by up to ∼ 52×. Not optimized plans still use vectorization and message batching (§5.2.3), otherwise the cost of secret sharing is prohibitive.

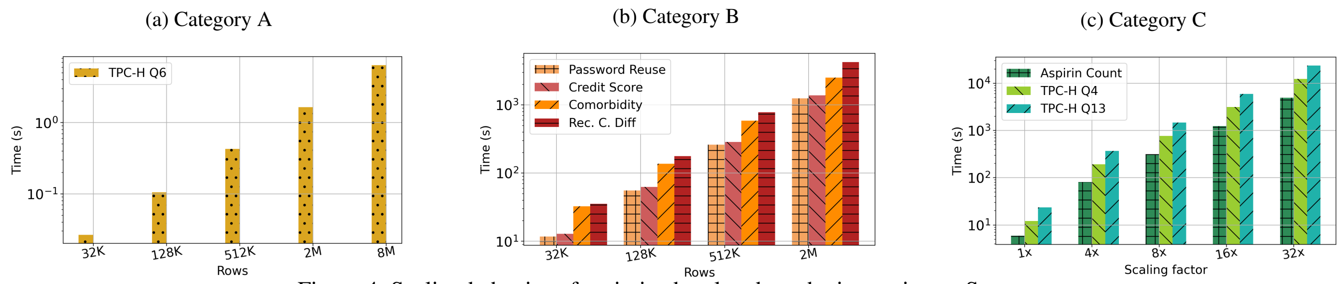


Figure 4: Scaling behavior of optimized real and synthetic queries on SECRECY

significant discrepancies in a particular year. In addition to the real-world queries, we use the TPC-H queries (**Q4**, **Q6**, **Q13**) [103] that have been used in SAQE [24]. Finally, to evaluate the performance gains from each optimization in isolation, we use **Q1**, **Q2**, **Q3** from §5.1-5.3.

Datasets. All experiments use randomly generated tables with 64-bit values. Note that SECRECY's MPC protocol assumes a fixed-size representation of shares that is implementation-specific and could be increased to any 2^k value. We also highlight that using random inputs is no different than using real data, as all operators are oblivious and the data distribution does not affect the amount of computation or communication. Regardless if the input is real or random, parties compute on secret shares, which are by definition random. In all experiments we designate one party as the data owner who distributes shares and learns the results. SECRECY uses exactly three computing parties; therefore, the number of data owners and analysts does not affect query performance, only the cumulative input size does.

7.2 Benefits of query optimization

We compare the performance of 8 queries optimized by SECRECY with that of plans without the optimizations of §5. For a fair comparison, we implement baseline plans using SECRECY's batched operators. Although this favors the baseline, the communication cost of MPC is otherwise prohibitive and queries cannot scale beyond a few hundred input rows. We execute each plan with $1K$ rows per input relation. For Q4 (resp. Q13), we use $1K$ rows for `LINEITEM` (resp. `ORDERS`) and maintain the size ratio with the other input relation as specified in the TPC-H benchmark. For *Comorbidity*, we use a cohort of 256 patients. We run this experiment on `AWS-LAN` and `MULTI-CLOUD` and present the results in Figure 3.

In the LAN setting, SECRECY achieves the highest speedups for *Recurrent C.Diff.*, *Aspirin Count*, and Q13, that is, 1142×, 121×, and 714× lower execution times, respectively. Optimized plans for these queries leverage join push-up (*Aspirin Count*), fusion (*Recurrent C.Diff.*), and join-aggregation decomposition (Q13). The optimized plans for *Comorbidity*, *Password Reuse*, Q4, and Q6 leverage dual and proactive sharing, achieving up to 52× speedup compared to the baseline. Finally, the *Credit Score* query leverages dual sharing which, in this case, provides a modest improvement. SECRECY achieves significant speedups in the wide area, too. The performance improvement is higher for *Comorbidity*, Q4, and Q13 in the multi-cloud setting, as these queries leverage optimizations that primarily reduce the synchronization cost. We evaluate the benefit of individual optimizations in §7.4.

7.3 Performance on real and synthetic queries

We now run the optimized plans with increasing input sizes in `AWS-LAN` and report total execution time. For these experiments, we group queries into three categories of increasing complexity. *Category A* includes queries with selections and global aggregations, *Category B* includes queries with select and group-by or distinct operators, and *Category C* includes queries with select, group-by and (semi-)join operators. Figure 4 presents the results.

Q6 in *Category A* consists of five selections and a global aggregation. It requires minimal communication that is independent of the input relation cardinality. As a result, it scales comfortably to large inputs and takes ∼ $6s$ for $8M$ rows.

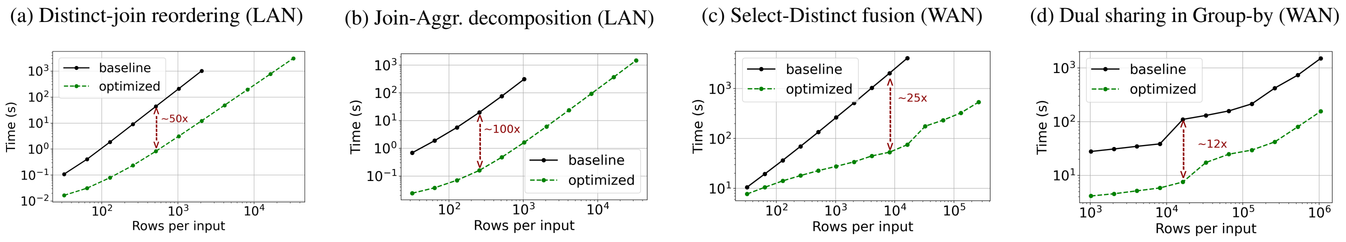


Figure 5: Performance improvement of individual optimizations applied by the SECRECY planner

Queries in *Category B* scale to millions of input rows as well. The cost of these queries is dominated by the oblivious group-by and distinct operators. At $2M$ rows, *Recurrent C.Diff.* completes in $\sim 1.2h$ and *Password Reuse* in $\sim 20min$.

The cost of queries in *Category C* is dominated by joins and semi-joins. The size ratio between the two inputs of each query is different: for Q4 and Q13, we use the ratio specified in the TPC-H benchmark whereas, for *Aspirin Count*, we use inputs of equal size. In Figure 4c, Scaling factor 1× corresponds to $1K$ rows for the small input. As we increase the input sizes, we always keep their ratio fixed. At scaling factor 32×, the most expensive query is Q13, which is optimized with join-aggregation decomposition and takes $\sim 6.5h$ on $295K$ rows. At the same scaling factor, Q4 completes in $\sim 3.4h$ on $164K$ rows, and *Aspirin Count* in $\sim 1.3h$.

While MPC protocols remain expensive for real-time queries, our results demonstrate that offline collaborative analytics on millions of records entirely under MPC are viable.

7.4 Micro-benchmarks

We now use the queries of §5 (**Q1**, **Q2**, **Q3**) to evaluate the impact of SECRECY's optimizations in isolation. We run each query with and without the particular optimization and measure total execution time. Distinct-join reordering and join-aggregation decomposition primarily reduce the operation cost and we evaluate them in `AWS-LAN`. Fusion and dual sharing reduce the synchronization cost and we evaluate them in `AWS-WAN`. Figure 5 shows the results.

Distinct-Join reordering. The optimized plan of Q1 pushes the `JOIN` after `DISTINCT` and, thus, only sorts a relation of n rows instead of n^2. Figure 5a shows that the optimized plan is up to 50× faster than the baseline, which runs out of memory for even modest input sizes.

Join-Aggregation decomposition. The baseline plan of Q2 materializes the result of the join and then applies the grouping and aggregation. Instead, the optimized plan decomposes the aggregation in two phases (cf. §5.1.3). As shown in Figure 5b, this optimization provides 100× lower execution time than that of the baseline plan. Further, the baseline plan runs out of memory for inputs larger than $1K$ rows.

Operator fusion. The baseline plan of Q3 applies the oblivious selection before `DISTINCT`, while the optimized plan fuses the two operators and performs the `DISTINCT` computation in bulk (cf. §5.2.2). Figure 5c shows that this optimization provides more than 25× speedup for large inputs and allows the query to scale to much larger inputs.

Dual sharing. We also evaluate SECRECY's ability to switch between arithmetic and boolean sharing to reduce communication costs for certain operations. For this experiment, we compare the run-time of the optimized `GROUP-BY-COUNT` operator (cf. §5.3) to that of a baseline operator that uses boolean sharing only and, hence, relies on the ripple-carry adder to compute the `COUNT`. Figure 5d plots the results. The baseline operator is 10× slower than the optimized one, as it requires 64 additional rounds of communication per input row.

	Comorbidity	Recurrent C. Diff.	Aspirin Count
SMCQL	91s	358s	365s
SECRECY	0.083s	0.092s	0.171s

Table 4: SMCQL and SECRECY execution times in LAN for the three medical queries from [22] on 25 tuples per input relation.

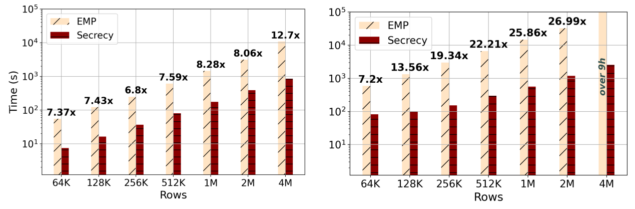


Figure 6: Performance comparison of the oblivious sort operator on EMP and SECRECY in LAN (left) and WAN (right).

7.5 Comparison with other MPC frameworks

Existing 3-party frameworks [62] are either proprietary, e.g. [27], or they only support specific operators, such as unique-key joins [82, 99], that cannot be used for any of the queries we consider. We stress that EMP and SMCQL use 2-party garbled circuit protocols that are not directly comparable with SECRECY's. The purpose of these experiments is to showcase the end-to-end performance of the available solutions for relational MPC and not to compare the underlying protocols.

Comparison with SMCQL. In the first set of experiments, we aim to reproduce the results presented in SMCQL [22, Fig. 7] on our experimental setup. We run the three medical queries on SMCQL and SECRECY, using a sample of 25 rows per data owner (50 in total), and present the results in Table 4. We use the plans and default configuration of protected and public attributes, as in the SMCQL project repository. SECRECY is over 1000× faster than SMCQL in all queries.

Comparison with EMP. EMP is a general-purpose MPC framework and does not provide relational operators or query planning. Nevertheless, we include a comparison with EMP because it is the MPC library underpinning several non-open-source relational frameworks (e.g., Shrinkwrap, SAQE, and Senate). For these experiments, we use the oblivious sort operation from the EMP repository [108] that has the same asymptotic complexity with the respective SECRECY sort. Figure 6 shows the results in `AWS-LAN` and `AWS-WAN` for input sizes ranging from $64K$ to $4M$ rows. The performance gap between SECRECY and EMP is significant. SECRECY is up to 12.7× faster in LAN ($\sim 3h$ vs $14min$ for $4M$ input rows). In the WAN setting, SECRECY sorts $4M$ rows in $42min$, while EMP could not complete the computation within $9h$.

8 Related Work

Relational MPC systems. We distinguish two lines of work in this space that are often combined. The first line targets peer-to-peer deployments and reduces multi-party computation by pushing parts of the query to data owners (for plaintext evaluation) or executing the protocol within subsets of the computing parties [13, 22, 44, 95, 105]. The second line includes systems that trade off MPC performance with controlled information leakage [23, 24, 64, 105, 111]. We summarize each system's preconditions and guarantees in Table 5 and provide more details in Appendix C. Function secret sharing in Splinter [106] allows for private queries on public data, which is the opposite to our goal.

Our approach has several advantages over, and is also complementary with, many of the prior techniques. SECRECY's optimizations are agnostic to data ownership and retain the full security guarantees of MPC, merely optimizing its execution. More importantly, this work provides a strong foundation for a unified query optimization framework that can accommodate multi-cloud, peer-to-peer, and hybrid deployments. Prior techniques can be ported into SECRECY by plugging in appropriate cost functions and query transformation rules. For example, pushing parts of the query to data owners, as in Conclave [105], can be done via transformation rules that introduce plaintext operators with certain placement constraints.

Enclave-based approaches. In this line of work, parties process the plaintext data within a physically protected environment. Enclave-based approaches aim to minimize RAM requirements, pad intermediate results, and hide access patterns in untrusted storage. The works by Agrawal et al. [14] and Arasu et al. [19] focus on database queries in this setting. More recent systems such as ObliDB [52], Opaque [116], StealthDB [104], and OCQ [49] rely on Intel's SGX.

Enclave-based systems typically achieve better performance than MPC systems but require different trust assumptions and are susceptible to attacks [33, 35, 36, 59, 74, 75, 107, 112]. Some of these threats can be ameliorated using oblivious operators within the enclave. Our logical optimizations from §5.1 could also be applied in this setting to reduce the number of operations and memory requirements.

System optimizations for MPC. Improving the performance of secure computation via system optimizations is an active research topic. MAGE [73] proposed an interesting technique to reduce the inherent memory overhead of homomorphic encryption and garbled circuits (cf. §3). As SECRECY relies on secret sharing, its memory footprint is small. Instead, secret sharing incurs a higher communication cost, which we amortize using vectorization and message batching (§5.2.3). MPC performance can be further improved by offloading secure primitives to hardware accelerators [54, 55, 79, 102]. Most works in this space focus on ML workloads but similar techniques could also be applied to relational operators.

MPC operators, algorithms, and cost models. Various related works focus on standalone oblivious relational operators, e.g. building group-by from oblivious sort [66], equi-joins [15, 72, 82, 93], or common aggregations [45, 51]. SECRECY is driven by real-world applications that typically require oblivious evaluation of queries with multiple operators. Motivated by similar needs, Wang et al. [110] presented a secure version of the Yannakakis' algorithm, while Ion et al. [65] and Buddhavarapu et al. [34] studied unique-key joins followed by simple aggregations. These works do not provide general cost-based MPC query optimization and they operate in the peer-to-peer setting, where data owners participate in the protocol execution using trusted resources. Recently, CostCO [53] did some nice work on modeling the cost of general MPC programs. Our cost model focuses on relational operators and is tightly integrated with the query planner.

Encrypted DBs. Existing practical solutions in secure database outsourcing [56] operate in a client-server setting and reveal or "leak" information to the database server. Systems based on property-based encryption like CryptDB [96] offer full SQL support and legacy compliance, but each query reveals information that can be used in reconstruction attacks [37,58,60,61,69,78,80,84]. Systems based on structural encryption [38, 67, 88, 94, 115] provide semantic security that does not eliminate access pattern leaks. SDB [64, 111] uses secret sharing but leaks information to the server whereas Cipherbase [18] relies on a trusted machine. These systems support only one data owner and it would require public-key encryption to evaluate queries that span multiple datasets [31].

Differential Privacy (DP). Systems like DJoin [83], DStress [87], and others [29, 43, 63] use DP to ensure that the out-

Framework	MPC Protocol	Information Leakage	Trusted Party	Query Execution	Main Optimization Objective	Optimization Conditions
Conclave [105]	Secret Sharing / Garbled Circuits	Controlled (Hybrid operators)	Yes	Hybrid	Minimize the use of secure computation	1. Data owners serve as computing parties 2. Data owners provide privacy annotations 3. There exists an additional trusted party
SMCQL [22]	Garbled Circuits / ORAM	No	No[1]	Hybrid	Minimize the use of secure computation	1. Data owners serve as computing parties 2. Data owners provide privacy annotations 3. There exists an honest broker
Shrinkwrap [23]	Garbled Circuits / ORAM	Controlled (Diff. Privacy)	No	Hybrid	Calibrate padding of intermediate results	1. Data owners serve as computing parties 2. Data owners provide privacy annotations and intermediate result sensitivities
SAQE [24]	Garbled Circuits	Controlled (Diff. Privacy)	No	Hybrid	Choose sampling rate for approximate answers	1. Data owners serve as computing parties 2. Data owners provide privacy annotations and differential privacy budgets
Senate [95] [2]	Garbled Circuits	No	No	Hybrid	Reduce joint computation to subsets of parties	1. Data owners serve as computing parties 2. Input or intermediate relations are owned by subsets of the computing parties
SDB [64, 111] [3]	Secret Sharing	Yes (operator dependent)	No	Hybrid	Reduce data encryption and decryption costs	1. Data owner serves as computing party 2. Data owner provides privacy annotations
SECRECY	Rep. Secret Sharing	No	No	End-to-end under MPC	Reduce MPC costs (§ 2.3 and § 4-5)	None

[1] *SMCQL relies on an honest broker that may see protected data in the clear during query evaluation* [22, § 5.1].
[2] *Senate provides security against malicious parties whereas all other systems adopt a semi-honest model.*
[3] *SDB adopts a typical DBaaS model with one data owner and does not support collaborative analytics.*

Table 5: Summary of MPC-based systems for relational analytics. Hybrid execution splits the query plan into a plaintext part (executed by the data owners) and an oblivious part (executed under MPC) and requires data owners to participate in the computation using trusted resources. The rest of the optimizations supported by each system are applicable under one or more of the listed conditions in the rightmost column.

put of a query reveals little about any one input record. This property is independent of (yet symbiotic with) MPC's guarantee that the act of computing the query reveals no more than what may be inferred from its output. The SECRECY primitives from §3.2 can express arbitrary computations and could also be used to add DP noise under MPC. We leave this as future work.

9 Conclusions

This work presents SECRECY, a new system for efficient secure analytics in the cloud with no information leakage. SECRECY can enable new data markets and socially-beneficial data analyses while protecting private data. Our results show that logical optimizations coupled with careful system design can make MPC practical for complex analytics on millions of data records. In the future, we plan to extend SECRECY with multi-objective query optimization that considers cloud fees, data-parallelism via oblivious hashing (e.g. [91, 92]), and support for malicious-secure MPC (e.g., [16, 71]).

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and James Mickens for shepherding the paper. We also thank Jonathan Appavoo, Orran Krieger, Ran Canetti, Leo Reyzin, Azer Bestavros, Kinan Dak Albab, and Ben Getchell for their comments on an early version of this work, Eric Chen for automating the SECRECY deployment, and the MOC Alliance for providing access to their cloud. This work has been partially supported by a Red Hat Collaboratory grant (No. 2022-01-RH02) and a Hariri Institute Focused Research Project Award. M. Varia's work is supported by the DARPA SIEVE program under Agreement No. HR00112020021 and the National Science Foundation under Grants No. 1414119, 1718135, 1801564, and 1931714.

References

[1] Cape Privacy. https://capeprivacy.com. [Online; accessed April 2022].

[2] General Data Protection Regulation (GDPR). https://gdpr.eu/tag/gdpr/. [Online; accessed April 2022].

[3] Health Insurance Portability and Accountability Act (HIPAA). https://www.cdc.gov/phlp/publications/topic/hipaa.html. [Online; accessed April 2022].

[4] HElib. https://github.com/homenc/HElib. [Online; accessed April 2022].

[5] Message Passing Interface (MPI). https://www.mcs.anl.gov/research/projects/mpi/standard.html. [Online; accessed April 2022].

[6] PALISADE. https://gitlab.com/palisade/palisade-release. [Online; accessed April 2022].

[7] Red Hat Ansible Automation Platform. https://www.redhat.com/en/technologies/management/ansible. [Online; accessed April 2022].

[8] SEAL. https://github.com/Microsoft/SEAL. [Online; accessed April 2022].

[9] Secrecy Github Repository. https://github.com/CASP-Systems-BU/Secrecy. [Online; accessed April 2022].

[10] SoK: General-Purpose Compilers for Secure Multi-party Computation. https://github.com/MPC-SoK/frameworks/blob/master/emp/sh_test/test/xtabs.cpp. [Online; accessed April 2022].

[11] The Carbyne Stack: Cloud Native Secure Multiparty Computation. https://carbynestack.io. [Online; accessed April 2022].

[12] The Hyrise Project: C++ SQL Parser. https://github.com/hyrise/sql-parser. [Online; accessed April 2022].

[13] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep a secret: A distributed architecture for secure database services. In *The Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, 2005.

[14] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, and Yaping Li. Sovereign joins. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 26, USA, 2006. IEEE Computer Society.

[15] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 86–97, New York, NY, USA, 2003. Association for Computing Machinery.

[16] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority mpc for malicious adversaries – breaking the 1 billion-gate per second barrier. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, pages 843–862, May 2017.

[17] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 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 (CCS)*, pages 805–817, October 2016.

[18] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*, January 2013.

[19] Arvind Arasu and Raghav Kaushik. Oblivious query processing. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 26–37. OpenProceedings.org, 2014.

[20] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 61(12):1749–1771, 2018.

[21] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex J. Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard W. Ryan. RAMPARTS: A programmer-friendly system for building homomorphic encryption applications. In *WAHC@CCS*, pages 57–68. ACM, 2019.

[22] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. SMCQL: secure query processing for private data networks. *Proc. VLDB Endow.*, 10(6):673–684, 2017.

[23] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. Shrinkwrap: efficient SQL query processing in differentially private data federations. *Proceedings of the VLDB Endowment*, 12(3):307–320, 2018.

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

[25] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography*, volume 8975 of *Lecture Notes in Computer Science*, pages 227–234. Springer, 2015.

[26] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and Taxes: a Privacy-Preserving Study Using Secure Computation. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2016(3):117–135, 2016.

[27] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[28] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[29] Jonas Böhler and Florian Kerschbaum. Secure sublinear time differentially private median computation. In *NDSS*. The Internet Society, 2020.

[30] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *ACM Conference on Computer and Communications Security*, pages 1175–1191. ACM.

[31] Christoph Bösch, Pieter H. Hartel, Willem Jonker, and Andreas Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys*, 47(2):18:1–18:51, 2014.

[32] Boston Women's Workforce Council (BWWC). Gender/racial pay gap in boston by the numbers. https://thebwwc.org, 2021.

[33] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[34] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Report 2020/599, 2020. https://eprint.iacr.org/2020/599.

[35] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 991–1008. USENIX Association, 2018.

[36] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

[37] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 668–679, New York, NY, USA, 2015. Association for Computing Machinery.

[38] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*. The Internet Society, 2014.

[39] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: high throughput 3pc over rings with application to secure prediction. In *CCSW@CCS*, pages 81–92. ACM, 2019.

[40] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS*. The Internet Society, 2020.

[41] Surajit Chaudhuri. An overview of query optimization in relational systems. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43. ACM Press, 1998.

[42] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 354–366. Morgan Kaufmann, 1994.

[43] Albert Cheu, Adam D. Smith, Jonathan R. Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *EUROCRYPT (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 375–403. Springer, 2019.

[44] Sherman S. M. Chow, Jie-Han Lee, and Lakshminarayanan Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*. The Internet Society, 2009.

[45] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 259–282, Boston, Massachusetts, USA, 2017. USENIX Association.

[46] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium*, pages 2183–2200. USENIX Association, 2021.

[47] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In *Financial Cryptography*, volume 9603 of *Lecture Notes in Computer Science*, pages 169–187. Springer, 2016.

[48] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[49] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *EuroSys*, pages 39:1–39:17. ACM, 2020.

[50] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[51] F. Emekci, D. Agrawal, A. E. Abbadi, and A. Gulbeden. Privacy preserving query processing using third parties. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 27–27, 2006.

[52] Saba Eskandarian and Matei Zaharia. Oblidb: oblivious query processing for secure databases. *Proceedings of the VLDB Endowment*, 13(2):169–183, 2019.

[53] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. CostCO: An automatic cost modeling framework for secure multi-party computation. In *IEEE EuroS&P 22*, 2022.

[54] Xin Fang, Stratis Ioannidis, and Miriam Leeser. SIFO: secure computational infrastructure using FPGA overlays. *Int. J. Reconfigurable Comput.*, 2019:1439763:1–1439763:18, 2019.

[55] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. Faster maliciously secure two-party computation using the GPU. In *SCN*, volume 8642 of *Lecture Notes in Computer Science*, pages 358–379. Springer, 2014.

[56] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. Sok: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 172–191. IEEE Computer Society, 2017.

[57] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[58] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In Pierangela Samarati, Mohammad S. Obaidat, and Enrique Cabello, editors, *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT, Madrid, Spain, July 24-26, 2017*, pages 200–211. SciTePress, 2017.

[59] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, pages 315–331. ACM, 2018.

[61] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1353–1364, New York, NY, USA, 2016. Association for Computing Machinery.

[62] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General purpose compilers for secure multi-party computation. In *IEEE Symposium on Security and Privacy*, pages 1220–1237. IEEE, 2019.

[63] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *ACM Conference on Computer and Communications Security*, pages 1389–1406. ACM, 2017.

[64] Zhian He, Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, Siu Ming Yiu, and Eric Lo. Sdb: A secure query processing system with data interoperability. *Proceedings of the VLDB Endowment*, 8(12):1876–1879, 2015.

[65] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. On deploying secure computing commercially: Private intersection-sum protocols and their business applications. *IACR Cryptology ePrint Archive*, 2019:723, 2019.

[66] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. Secure multi-party sorting and applications. *IACR Cryptol. ePrint Arch.*, 2011:122, 2011.

[67] Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 149–180, Cham, 2018. Springer International Publishing.

[68] Randy Howard Katz and Gaetano Borriello. *Contemporary logic design (2. ed.)*. Pearson Education, 2005.

[69] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1329–1340, New York, NY, USA, 2016. Association for Computing Machinery.

[70] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

[71] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: super-fast and robust privacy-preserving machine learning. In *30th USENIX Security Symposium*. USENIX Association, 2021.

[72] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *Proc. VLDB Endow.*, 13(11):2132–2145, 2020.

[73] Sam Kumar, David E. Culler, and Raluca Ada Popa. MAGE: Nearly zero-cost virtual memory for secure computation. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 367–385. USENIX Association, July 2021.

[74] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 487–504. USENIX Association, 2020.

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

[76] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[77] Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, December 2020.

[78] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 265:176–188, may 2014.

[79] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2505–2522, 2020.

[80] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 279–296. IEEE Computer Society, 2018.

[81] Payman Mohassel and Peter Rindal. Aby^3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 35–52, New York, NY, USA, 2018. Association for Computing Machinery.

[82] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and PSI for secret shared data. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1271–1287. ACM, 2020.

[83] Arjun Narayan and Andreas Haeberlen. Djoin: Differentially private join queries over distributed databases. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 149–162, October 2012.

[84] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[85] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 334–348. IEEE Computer Society, 2013.

[86] Takashi Nishide and Kazuo Ohta. Constant-round multiparty computation for interval test, equality test, and comparison. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 90-A(5):960–968, 2007.

[87] Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. Dstress: Efficient differentially private computations on distributed data. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 560–574, 2017.

[88] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374, 2014.

[89] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: improved mixed-protocol secure two-party computation. pages 2165–2182, 2021.

[90] Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *NDSS*. The Internet Society, 2020.

[91] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security Symposium*, pages 515–530. USENIX Association, 2015.

[92] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 122–153. Springer, 2019.

[93] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 125–157. Springer, 2018.

[94] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An encrypted database using semantically secure encryption. *Proc. VLDB Endow.*, 12(11):1664–1678, July 2019.

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

[96] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, 2011.

[97] Anjana Rajan, Lucy Qin, David W. Archer, Dan Boneh, Tancrède Lepoint, and Mayank Varia. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In *COMPASS*, pages 49:1–49:4. ACM, 2018.

[98] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 2nd edition, 2000.

[99] Peter Rindal. https://github.com/ladnir/aby3, The ABY3 Framework for Machine Learning and Database Operations. [Online; accessed April 2022].

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

[101] Riivo Talviste. *Practical applications of secure multiparty computation*, chapter 12, pages 246–251. IOS Press, 2015.

[102] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. Cryptgpu: Fast privacy-preserving machine learning on the gpu. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2021.

[103] Transaction Processing Performance Council. TPC Benchmark H. http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf. [Online; accessed April 2022].

[104] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full SQL query support. *Proc. Priv. Enhancing Technol.*, 2019(3):370–388, 2019.

[105] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 3:1–3:18. ACM, 2019.

[106] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 299–313, USA, 2017. USENIX Association.

[107] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2421–2434, New York, NY, USA, 2017. Association for Computing Machinery.

[108] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[109] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, pages 21–37. ACM, 2017.

[110] Yilei Wang and Ke Yi. *Secure Yannakakis: Join-Aggregate Queries over Private Data*, pages 1969–1981. Association for Computing Machinery, New York, NY, USA, 2021.

[111] Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, and Siu Ming Yiu. Secure query processing with data interoperability in a cloud database environment. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1395–1406, 2014.

[112] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, USA, 2015. IEEE Computer Society.

[113] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 89–100. IEEE Computer Society, 1994.

[114] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, USA, 1986. IEEE Computer Society.

[115] Zheguang Zhao, Seny Kamara, Tarik Moataz, and Zdonik Stan. Encrypted databases: From theory to systems. In *Proceedings of the 11th Annual Conference on Innovative Data Systems Research*, 2021.

[116] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, 2017.

[117] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. Cerebro: A platform for multi-party cryptographic collaborative learning. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2723–2740. USENIX Association, 2021.

A ANALYTICAL COST MODEL

A.1 Oblivious SECRECY primitives

Boolean operations. As explained in §3.2, we consider that each single-bit operation on shares has unit cost, so the operation cost of both XOR and AND operations is $C_o(\texttt{XOR}) = C_o(\texttt{AND}) = \ell$, where ℓ is the length of the share representation in bits. Recall that the synchronization cost of XOR is $C_s(\texttt{XOR}) = 0$ whereas the synchronization cost of AND is $C_s(\texttt{AND}) = 1$ round of communication. We further explain these costs below.

In SECRECY, each party starts with two shares of the input secrets s, t and ends up with two shares of the output. Initially, P_1 has s_1, s_2, t_1, t_2 whereas P_2 has s_2, s_3, t_2, t_3, and P_3 has s_3, s_1, t_3, t_1. Observe that $s \oplus t = (s_1 \oplus s_2 \oplus s_3) \oplus (t_1 \oplus t_2 \oplus t_3) = (s_1 \oplus t_1) \oplus (s_2 \oplus t_2) \oplus (s_3 \oplus t_3)$. Each parenthesis corresponds to a share of $s \oplus t$ and each party can compute two out of the three shares by simply XORing its input shares.

Logical AND is a bit more complex. Observe that $st = (s_1 \oplus s_2 \oplus s_3) \wedge (t_1 \oplus t_2 \oplus t_3)$. After distributing the AND over the XOR and doing some rearrangement we have $st = (s_1t_1 \oplus s_1t_2 \oplus s_2t_1) \oplus (s_2t_2 \oplus s_2t_3 \oplus s_3t_2) \oplus (s_3t_3 \oplus s_3t_1 \oplus s_1t_3)$. Again, each parenthesis corresponds to a share of st. Using its input shares, each party can locally compute one of these shares. The parties then XOR this share with a fresh sharing of the number 0 (which is created locally) so that the final share is uniformly distributed [17]. In the end, each party sends the computed share to its successor on the ring (clockwise) so that all parties end up with two shares of st. Logical OR and NOT are based on the XOR and AND primitives.

Equality/Inequality. Using these boolean operations, parties can jointly compute $s \overset{?}{=} t$ (resp. $s \overset{?}{<} t$) by computing a sharing of $s \oplus t$ and then taking the oblivious boolean-AND of each of the bits of this string (resp., taking the value of s_i at the first bit i in which the two strings differ). As a result, taking the equality of ℓ-bit strings requires $C_o(\text{eq}) = 2\ell - 1$ operations (namely, ℓ XORs plus $\ell - 1$ ANDs) and $C_s(\text{eq}) = \lceil \log \ell \rceil$ rounds. Similarly, inequality comparison has $C_o(\text{ineq}) = 4\ell - 3$ and $C_s(\text{ineq}) = \lceil \log(\ell + 1) \rceil$. As special cases, $s < 0$ requires no communication, and equality with a public constant $s{=}c$ can also be done locally provided that the data owners have secret-shared the results of $s - c$ and $c - s$ [70].

Compare-and-swap. The parties can calculate the min and max of two strings. Setting $b = (s{<}t)$, we can use a multiplexer to compute $s' = \min\{s, t\} = bs \oplus (1 \oplus b)t$ and $t' = \max\{s, t\} = (1 \oplus b)s \oplus bt$. Evaluating these formulas requires 6 more operations and 1 more synchronization round beyond the cost of the oblivious inequality.

Sort and shuffle. Given an array of n secret-shared strings, each of length ℓ, oblivious sort in SECRECY is based on a bitonic sorter that comprises $\log n \cdot (\log n + 1)/2$ stages and performs $n/2$ independent compare-and-swap operators in each stage. Hence, sorting has operational cost $C_o(\text{sort}_n) = \frac{1}{4} n \log n \cdot (\log n + 1) \cdot (C_o(\text{ineq}) + 6)$ and synchronization cost $C_s(\text{sort}_n) = \frac{1}{2} \log n \cdot (\log n + 1) \cdot (C_s(\text{ineq}) + 1)$. We can obliviously shuffle values in a similar fashion: each party appends an attribute that is populated with locally generated random values, sorts the values on this attribute, and then discards it.

Boolean addition. Given boolean-shared integers s and t, computing the boolean share of $s + t$ using a ripple-carry adder [68] can be done with $C_o(\texttt{RCA}) = 5\ell - 3$ operations in $C_s(\texttt{RCA}) = \ell$ rounds.

Arithmetic operations. Arithmetic addition and multiplication work similarly to XOR and AND respectively (see above). Scalar multiplication $c \cdot u$, where c is a public constant, does not require communication.

Conversion. We can convert between additive and boolean sharings [50, 81, 89] by securely computing all of the XOR and AND gates in a ripple-carry adder. Single-bit conversion can be done in two rounds with the simple protocol that is also used in CrypTen [70].

A.2 Oblivious SECRECY operators

Let R, S, and T be relations with cardinalities $|R|$, $|S|$, and $|T|$ respectively. Let also $t[a_i]$ be the value of attribute a_i in tuple t. To simplify the presentation, we describe each operator based on the logical (i.e., secret) relations and not the random shares distributed across parties. That is, when we say that "*an operator is applied to a relation R and defines another relation T*", in practice this means that each party begins with shares of R, performs some MPC operations on the shares, and ends up with shares of T.

PROJECT. Oblivious projection has the same semantics as its plaintext counterpart. The operation and synchronization costs of oblivious `PROJECT` are both zero since each party can locally disregard the shares corresponding to the filtered attributes.

SELECT. An oblivious selection with predicate φ on a relation R defines a new relation:

$$T = \{t \cup \{\varphi(t)\} \mid t \in R\}$$

with the same cardinality as R, i.e. $|T| = |R|$, and one more single-bit attribute for each tuple $t \in R$ that contains ϕ's result when applied to t. This bit denotes whether t is included in the output relation T and is securely computed under MPC so that its true value remains hidden (i.e., secret-shared) from the computing parties. Note that, in contrast to a typical selection in the clear, oblivious selection defines a relation with the same cardinality as the input, i.e., it does not remove tuples from R so that the true size of T is kept secret.

Costs: The operation cost of `SELECT` is $C_o(\sigma_\phi(R)) = C_o(\phi(t)) \cdot |R|, \quad t \in R$, where $C_o(\phi(t))$ is the operation cost of evaluating ϕ on a single tuple $t \in R$. Since predicate evaluation can be performed independently for each tuple in R, the total number of rounds to perform the `SELECT` equals the number of rounds to evaluate the selection predicate on a single tuple, i.e., $C_s(\sigma_\phi(R)) = C_s(\phi(t)),\ t \in R$.

Both $C_o(\phi(t))$ and $C_s(\phi(t))$ are independent of the actual t contents: they only depend on ϕ's syntax and the lengths of the attributes used in ϕ. In SECRECY, a predicate ϕ can be an arbitrary logical expression with atoms that may also include arithmetic expressions $(+, \times, =, >, <, \neq, \geq, \leq)$ and is constructed using the primitives of §A.1. Consider the example predicate $\phi :=$ `age>30 AND age<40` that requires `AND`ing the results of two oblivious inequalities under MPC. Based on the costs of primitive operations, we have: $C_o(\phi(t)) = 2C_o(\text{ineq}) + C_o(\text{AND})$ and $C_s(\phi(t)) = C_s(\text{ineq}) + 1$. In §5.3, we described a technique we use in SECRECY that can reduce selections to local operations (with $C_s = 0$).

JOIN. An oblivious θ-join between two relations R and S, denoted with $R \bowtie_\theta S$, defines a new relation:

$$T = \{(t \cup t' \cup \{\theta(t,t')\}) \mid t \in R \wedge t' \in S\}$$

where $t \cup t'$ is a new tuple that contains all attributes of $t \in R$ along with all attributes of $t' \in S$, and $\theta(t,t')$ is θ's result when applied to the pair of tuples (t,t'). T is the cartesian product of the input relations $(R \times S)$, where each tuple is augmented with a (secret-shared) bit denoting whether the tuple t "matches" with tuple t' according to θ. We emphasize that our focus in this work is on general-purpose oblivious joins that can support arbitrary predicates; there also exist special cases of oblivious join algorithms, e.g., primary- and foreign-key equi-joins with lower asymptotic complexity [15, 72, 82, 93] or compositions of equi-joins with specific operators [34] that could be added to SECRECY if desired.

Costs: The general oblivious `JOIN` requires a nested-loop over the input relations to check all possible pairs, so its operation cost is $C_o(R \bowtie_\theta S) = C_o(\theta(t,t')) \cdot |R| \cdot |S|,\ t \in R, t' \in S$. However, the total number of communication rounds to evaluate the `JOIN` is independent of the input cardinality; it only depends on the join predicate θ, i.e., $C_s(R \bowtie_\theta S) = C_s(\theta(t,t')), t \in R, t' \in S$. For example, a range join $R \bowtie_{a<b} S$ has $C_o(R \bowtie_{a<b} S) = 2|R| \cdot |S| \cdot C_o(\text{ineq})$ and $C_s(R \bowtie_{a<b} S) = C_s(\text{ineq})$. The constant asymptotic complexity in number of rounds with respect to the input cardinality holds for *any* θ-join. Join predicates in SECRECY can be arbitrary expressions whose cost is computed as explained above for selection predicates.

SEMI-JOIN. An oblivious (left) semi-join between two relations R and S on a predicate θ, denoted with $R \ltimes_\theta S$, defines a new relation:

$$T = \{(t \cup \{\bigvee_{\forall t' \in S} \theta(t,t')\}) \mid t \in R\}$$

with the same cardinality as R, i.e. $|T| = |R|$, and one more attribute that stores the result of the formula $f(\theta, t, S) = \bigvee_{\forall t' \in S} \theta(t,t'),\ t \in R$ indicating whether the tuple in R "matches" any tuple in S.

Costs: The operation cost of the general oblivious `SEMI-JOIN` is $C_o(R \ltimes_\theta S) = C_o(f(\theta,t,S)) \cdot |R| = C_o(\theta(t,t')) \cdot |R| \cdot |S| + |R| \cdot (|S| - 1),\ t \in R, t' \in S$. The formula $f(\theta,t,S)$ can be evaluated independently for each tuple $t \in R$ using a binary tree of `OR` operations, therefore, the synchronization cost of the semi-join is $C_s(R \ltimes_\theta S) = C_s(\theta(t,t')) + \lceil \log |S| \rceil, t \in R, t' \in S$ (i.e., independent of $|R|$).

ORDER-BY. Oblivious order-by on attribute a_k has the same semantics as the non-oblivious operator. Hereafter, sorting a relation R with m attributes on ascending (resp. descending) order of an attribute $a_k, 1 \leq k \leq m$, is denoted as $s_{\uparrow a_k}(R) = T$ (resp. $s_{\downarrow a_k}(R) = T$). We define order-by on multiple attributes using the standard semantics. For example, sorting a relation R first on attribute a_k (ascending) and then on a_n (descending) is denoted as $s_{\uparrow a_k \downarrow a_n}(R)$. An order-by operator is often followed by a `LIMIT` that defines the number of tuples the operator must output.

Costs: Oblivious `ORDER-BY` in SECRECY relies on a bitonic sorter of §A.1 that internally uses an oblivious multiplexer. Hence, the operation and synchronization costs are $C_o(s_{\uparrow a}(R)) = C_o(\text{sort}_{|R|})$ and $C_s(s_{\uparrow a}(R)) = C_s(\text{sort}_{|R|})$, as given in §A.1. In this case, the number of operations required by each oblivious multiplexing is linear to the number of attributes in the input relation, however, the total number of rounds depends only on the cardinality of the input. The analysis assumes one sorting attribute; adding more sorting attributes increases the number of operations and communication rounds in each comparison by a small constant factor.

GROUP-BY with aggregation. An oblivious group-by aggregation on a relation R with m attributes defines a new relation $T = \{f(t') \mid t' = t \cup \{a_g, a_v\},\ t \in R\}$ with the same cardinality as R, i.e. $|T| = |R|$, and two more attributes: a_g that stores the result of the aggregation, and a_v that denotes whether the tuple t is 'valid', i.e., included in the output. Let a_k be the group-by key and a_w the attribute whose values are aggregated. Let also $S = \left[t_1[a_w], t_2[a_w], ..., t_u[a_w]\right]$ be the list of values for attribute a_w for all tuples $t_1, t_2, ..., t_u \in R$ that belong to the same group, i.e., $t_1[a_k] = t_2[a_k] = ... = t_u[a_k]$, $1 \leq u \leq |R|$. The function f in T's definition above is defined as follows:

$$f(t_i) = \begin{cases} t_i[a_g] = agg(S), t_i[a_v] = 1, & i = u',\ 1 \leq u' \leq u \\ t_{inv}, & i \neq u',\ 1 \leq i \leq u \end{cases}$$

where t_{inv} is a tuple with $t_{inv}[a_v] = 0$ and the rest of the attributes set to a special reserved value, while $agg(S)$ is the aggregation function, e.g. `MIN`, `MAX`, `COUNT`, `SUM`, `AVG`, and is implemented using the primitives of §A.1. Put simply, oblivious aggregation sets the value of a_g for one tuple per group equal to the result of the aggregation for that group and updates (in-place) all other tuples with "garbage." This operation is followed by an oblivious shuffling to hide the group boundaries when opening the relations to the learner (and only if there is no subsequent shuffling in the query plan). Groups can be defined on multiple attributes using the standard semantics. Costs: The `GROUP-BY` operator $\gamma_{a_k}^{agg}(R)$ breaks into two phases: an oblivious sort on the group-by key(s) and an odd-even aggregation [66] applied to the sorted input. The odd-even aggregation performs $(|R|(\log|R|-1)+1)\cdot C_o(agg(t,t'))$ operations in $\log|R|\cdot C_s(agg(t,t'))$ rounds, where $C_o(agg(t,t'))$ and $C_s(agg(t,t'))$ are the operation and synchronization costs, respectively, of applying the aggregation function to a single pair of tuples $t, t' \in R$ (independent of $|R|$). Accounting for the initial sorting on the group-by keys, the total operation cost of the oblivious group-by is $C_o(\gamma_{a_k}^{agg}(R)) = C_o(s_{\uparrow a_k}(R)) + (|R|(\log|R| - 1) + 1)\cdot C_o(agg(t,t'))$. The total synchronization cost is $C_s(\gamma_{a_k}^{agg}(R)) = C_s(s_{\uparrow a_k}(R)) + \log|R|\cdot C_s(agg(t,t'))$. The analysis can be easily extended to multiple group-by keys.

DISTINCT. The oblivious distinct operator is a special case of group-by with aggregation, assuming that a_k is not the group-by key as before but the attribute where distinct is applied. For distinct, there is no a_g attribute and the function f is defined as follows:

$$f(t_i) = \begin{cases} t_i[a_v] = 1, & i = u', \ 1 \leq u' \leq u \\ t_i[a_v] = 0, & i \neq u', \ 1 \leq i \leq u \end{cases}$$

Distinct marks one tuple per group as 'valid' and the rest as 'invalid'.

Costs: The `DISTINCT` operator includes an oblivious sort on the distinct attribute(s) followed by a second phase where the operator compares adjacent tuples in the sorted input to set the distinct bit a_v. Setting the distinct bit for each tuple is independent from the rest of the tuples, so all distinct bit operations can be performed in bulk. The total operation cost $C_o(\delta a_k(R)) = C_o(s_{\uparrow a_k}(R)) + (|R|-1)\cdot C_o(\text{eq})$ and synchronization cost $C_s(\delta a_k(R)) = C_s(s_{\uparrow a_k}(R)) + C_o(\text{eq})$ of oblivious distinct are dominated by the oblivious sort.

MASK. Let t_{inv} be a tuple with all attributes set to a special reserved value. A mask operator with predicate p on a relation R defines a new relation $T = \{f(t) \mid t \in R\}$, where:

$$f(t) = \begin{cases} t, & p(t) = 0 \\ t_{inv}, & p(t) = 1 \end{cases}$$

Mask is used at the end of the query, just before opening the result to the learner, and only if there is no previous masking. The cost analysis of `MASK` is similar to that of `SELECT`.

Global aggregations. SECRECY also supports global aggregations without a group-by clause. The total operation cost of a global aggregation is $C_o(agg(R)) = C_o(agg(t,t'))\cdot(|R|-1)$, where $C_o(agg(t,t'))$ is the operation cost of applying the aggregation function to a single pair of tuples $t, t' \in R$. The total synchronization cost is $C_s(agg(R)) = C_s(agg(t,t'))\cdot\lceil\log|R|\rceil$, since the aggregation can be applied using a binary tree of function evaluations.

A.3 Composition of oblivious operators

Composing selections and joins. Recall that selections, joins, and semi-joins append a single-bit attribute to their input relation that indicates whether the tuple is included in the output. To compose a pair of such operators, we compute both single-bit attributes and take their conjunction under MPC. For example, for two selection operators σ_1 and σ_2 with predicates φ_1, φ_2, the composition $\sigma_2(\sigma_1(R))$ defines a new relation $T = \{t \cup \{e_c = \varphi_1(t) \wedge \varphi_2(t)\} \mid t \in R\}$. The cost of composition in this case is the cost of evaluating the expression $\varphi_1(t) \wedge \varphi_2(t)$ for each tuple in T. This includes $|T|$ independent boolean ANDs which can be evaluated in one round.

Composing distinct with other operators. Applying a selection or a (semi-)join to the result of `DISTINCT` requires one communication round to compute the conjunction of the selection or (semi-) join bit with the bit a_v generated by distinct. However, applying `DISTINCT` to the output of a selection, a (semi-)join or a group-by operator, requires some care. Consider the case where `DISTINCT` is applied to the output of a selection. Let a_ϕ be the attribute added by the selection and a_k be the distinct attribute. To set the distinct bit a_v at each tuple, we must make sure there are no other tuples with the same attribute a_k, with $a_\phi = 1$, and whose distinct bit a_v is already set. To do so, the distinct operator must process tuples *sequentially* and the composition itself requires n rounds, where n is the cardinality of the input. This results in a significant increase over the $O(\log^2 n)$ rounds required by distinct when applied to a base relation. Applying distinct to the output of a group-by or (semi-)join incurs a linear number of rounds for the same reason. In §5.2, we proposed an optimization that reduces the cost of these compositions to a logarithmic factor.

Composing group-by with other operators. To perform a group-by on the result of a selection or (semi-)join, the group-by operator must apply the aggregation function to all tuples in the same group that are also included in the output of the previous operator. Consider the case of applying group-by to a selection result. To identify the aforementioned tuples, we need to evaluate the formula:

$$b \leftarrow b \wedge t_i[a_\phi] \wedge t_j[a_\phi]$$

at each step of the group-by operator, where b is the bit that denotes whether the tuples t_i and t_j belong to the same group and a_ϕ is the selection bit. This formula includes two boolean ANDs that require two communication rounds. Applying group-by to the output of a (semi-)join has the same composition cost; in this case, we replace a_ϕ in the above formula with the (semi-)join attribute a_θ.

To apply a selection to the result of `GROUP-BY`, we must compute a boolean AND between the selection bit a_ϕ and the 'valid' bit a_v of each tuple generated by the group-by. The cost of composition in number of rounds is independent of the group-by result cardinality, as all boolean ANDs can be applied in bulk. The same holds when applying a (semi-)join to the output of group-by. Finally, composing two group-by operators has the same cost with applying `GROUP-BY` to the result of selection, as described above.

Composing order-by with other operators. Composing `ORDER- BY` with other operators is straight-forward. Applying an operator to the output of order-by has zero composition cost. The converse operation, applying `ORDER-BY` to the output of an operator, requires a few more boolean operations per oblivious compare-and-swap (due to the attribute/s appended by the previous operator), but does not incur additional communication rounds.

B SECURITY ANALYSIS

We have purposely designed SECRECY in a modular *black-box* fashion, with a hierarchy of *MPC protocol functionalities* $\longrightarrow$ *oblivious primitives* $\longrightarrow$ *relational operators* $\longrightarrow$ *optimizations*. This design choice provides two benefits: (i) immediate *inheritance* of all security guarantees provided by the underlying MPC protocol, and (ii) *flexibility* to support different protocols in the future that might have a different number of parties, threshold, and threat model.

Inheritance of security guarantees. SECRECY relies on a set of functionalities that must be provided by the MPC protocol. These functionalities enable parties to receive secret-shared inputs and return secret-shared outputs: (i) $\mathcal{F}_{\mathsf{add}}$ and $\mathcal{F}_{\mathsf{mult}}$ that add and multiply their inputs, (ii) $\mathcal{F}_{\mathsf{xor}}$ and $\mathcal{F}_{\mathsf{and}}$ that take boolean operations of their inputs, (iii) $\mathcal{F}_{\mathsf{a2b}}$ and $\mathcal{F}_{\mathsf{b2a}}$ that perform conversions between arithmetic and boolean representations, (iv) $\mathcal{F}_{\mathsf{eq}}$ and $\mathcal{F}_{\mathsf{cmp}}$ to compute the equality and comparison predicates (where the hardest step of the latter usually involves extracting the most significant bit of an arithmetic-shared value), and (v) $\mathcal{F}_{\mathsf{sh}}$ and $\mathcal{F}_{\mathsf{rec}}$ that allow external participants to secret-share data to and reconstruct data from the computing parties.

In this section, we argue that SECRECY retains the security guarantees provided by the underlying MPC protocol, or equivalently that it retains the security guarantees of these ideal functionalities. Our reasoning shows that SECRECY compiles each query into a sequence of calls to these functionalities that is *oblivious*, meaning that its control flow is independent of its input and all data remains hidden:

1. SECRECY calls the functionalities of the MPC protocol in a black-box manner. As a result, computing parties always operate on secret-shared data; only $\mathcal{F}_{\mathsf{rec}}$ provides any data in the clear (namely to the learner), and SECRECY only calls this functionality once at the end of the query execution.
2. The control flow of each relational operator (§A.2) is oblivious, i.e., data-independent. Concretely, `SELECT` and `PROJECT` always require a single pass over the input, (semi-)`JOIN`s require a nested for-loop over the two inputs, `ORDER-BY` is based on an oblivious sorting network, and `GROUP-BY` and `DISTINCT` consist of an `ORDER-BY` followed by an additional oblivious step (to apply the aggregation and identify the unique records, respectively).
3. SECRECY composes relational operators (§A.3) using the protocol functionalities (e.g., taking ANDs under MPC) within an oblivious linear scan over the output of the composition.
4. The logical and physical transformations of §5 rewrite the oblivious sequence of calls to the protocol functionalities into a new semantically equivalent sequence of calls that is also oblivious and has lower execution cost.

As a result, semi-honest security of the full SECRECY protocol follows by inspection of the ideal functionalities. Privacy is satisfied against all parties because none of the functionalities ever provides a (non-secret-shared) output to the data owners or computing parties, and only the final $\mathcal{F}_{\mathsf{rec}}$ provides an output to the analyst as desired. Correctness of the full protocol follows immediately from correctness of each individual functionality.

Generality of optimizations. The logical and physical query optimizations constructed in this work (§5.1-5.2) apply generally to any mixed-mode MPC protocol that supports the set of functionalities we describe above. This level of abstraction is commonly used by modern mixed-mode MPC protocols (e.g., [17, 39, 40, 46, 50, 71, 81, 89, 90]).

If providing malicious security, we require these functionalities to validate the shares of their inputs and outputs (e.g., using an information-theoretic MAC or replicated sharing), either immediately or with delayed validation before invoking $\mathcal{F}_{\mathsf{rec}}$. As a consequence, SECRECY satisfies correctness against the computing parties because input validation binds them to provide the output of the prior step as the input shares into the next functionality. Additionally, correctness against the data owners and analyst follow from the fact that, aside from the data owners' initial sharing through $\mathcal{F}_{\mathsf{sh}}$, none of the functionalities allow them to provide an input so they cannot influence the protocol execution.

As a result, the techniques from SECRECY can be applied to any N-party MPC protocol that provides semi-honest or malicious security against T adversarial parties. In particular, SECRECY can be instantiated with 2, 3, and 4-party secret sharing-based protocols that remain secure in the face of a malicious adversary who can deviate from the protocol arbitrarily (e.g., [46, 71, 89]), or with (authenticated) garbled circuit protocols [109, 114] combined with occasional conversions to arithmetic secret sharing [50,89] as needed. Protocols that provide the stronger cryptographic guarantee of robustness often do so by running several MPC executions both before and after evicting the malicious party, and by the same logic as above SECRECY even maintains the robust security of these protocols.

C SUMMARY OF MPC-BASED SYSTEMS FOR RELATIONAL ANALYTICS

Here we provide more details on existing systems for relational MPC summarized in Table 5. Hybrid execution splits the query plan into parts that can be evaluated in plaintext by a single party (the appropriate data owner) versus parts that inherently require multiple parties' data (executed under MPC). Therefore, hybrid execution is only feasible when data owners can compute part of the query on premise. SMCQL, SDB, and Conclave can further sidestep MPC when some attributes have been annotated as non-sensitive. Shrinkwrap and SAQE build on SMCQL to calibrate leakage based on user-provided privacy budgets, and Senate reduces joint computation when some relations are owned by subsets of the computing parties. This is common in peer-to-peer MPC but does not occur in a typical outsourced setting like the one of Figure 1, where all computing parties receive shares of the input data.

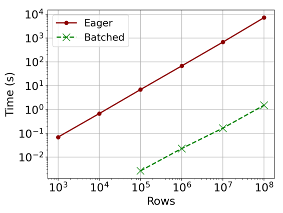
As shown in Table 5, Senate is the only relational system with support for malicious security. Due to its hybrid execution model, Senate requires additional steps to verify the integrity of local computations by the data owners (not to be confused with formal software verification). While SECRECY currently focuses on semi-honest security, the Araki et al. protocol [17] and subsequent mixed-mode ABY^3 protocols [81] can be extended to provide malicious security with low computational cost [16, 71]. By optimizing MPC rather than sidestepping it, a malicious-secure version of SECRECY would not impose any new restriction on the supported set of queries or the mixed-mode MPC protocol utilized.

D ADDITIONAL EXPERIMENTS

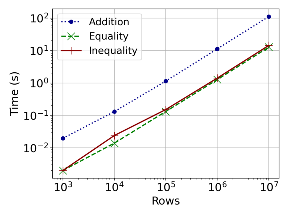
D.1 Performance of SECRECY primitives

Here we present a set of micro-benchmarks that evaluate the performance of SECRECY's MPC primitives in `AWS-LAN`.

Effect of message batching on communication latency. In the first experiment, we measure the latency of inter-party communication using two messaging strategies. Recall that, during a message exchange, each party sends one message to its successor and receives one message from its predecessor on the logical 'ring'. *Eager* exchanges data among parties as soon as they are generated, thus,. producing a large number of small messages. The *Batched* strategy, on the other hand, collects data into batches and exchanges them only when computation cannot otherwise make progress, thus, producing as few as possible, albeit large messages.

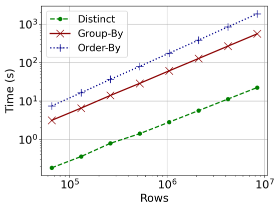


(a) Effect of message batching

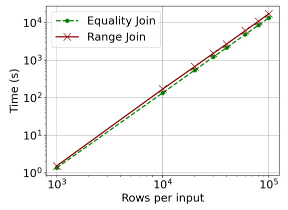


(b) Comparison and addition

Figure 7: Performance of oblivious SECRECY primitives



(a) Unary operators



(b) Join operators

Figure 8: Performance of oblivious SECRECY operators

We run this experiment with increasing data sizes and measure the total time from initiating the exchange until all parties complete the exchange. Figure 7a shows the results. We see that batching provides two to four orders of magnitude lower latency than eager messaging. Using batching in our experimental setup, parties can exchange $100M$ 64-bit data shares in $2s$. These results reflect the network performance in our cloud testbed. We expect better performance in dedicated clusters with high-speed networks and higher latencies if the computing parties communicate over the internet.

Performance of secure computation primitives. We now evaluate the performance of oblivious primitives that require communication among parties. These include equality, inequality, and addition with the ripple-carry adder. In Figure 7b we show the execution time of oblivious primitives as we increase the input size from $1K$ rows to $10M$ rows. All primitives scale well with the input size as they all depend on a constant number of communication rounds. Equality requires six rounds. Inequality requires seven rounds and more memory than equality. Boolean addition is not as computation-intensive as inequality, but requires a higher number of rounds (64).

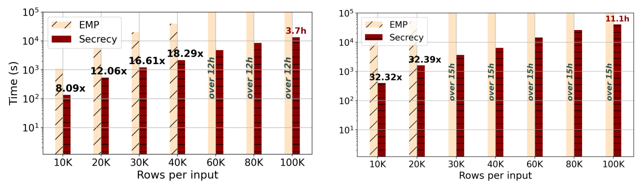


Figure 9: Performance comparison of an oblivious join operator on EMP and SECRECY in LAN (left) and WAN (right).

D.2 Performance of SECRECY operators

The next set of experiments evaluates the performance of SECRECY's relational operators. We apply `DISTINCT`, `GROUP-BY`, `ORDER-BY`, and `JOIN` (equality and range) to relations of increasing size and measure the total execution time per operator in `AWS-LAN`. We empirically verify the cost analysis of §A and show that our vectorized implementations are efficient and scale to millions of input rows with a single CPU thread. Figure 8 shows the results.

Unary operators. In Figure 8a, we plot the execution time of unary operators vs the input size. Recall from §A.2 that `DISTINCT` and `GROUP-BY` are both based on sorting and, thus, their cost includes the cost of `ORDER-BY` for unsorted inputs of the same cardinality. To shed more light on the performance of `DISTINCT` and `GROUP-BY`, Figure 8a only shows the execution time of their second phase, that is, after the input is sorted and, for `GROUP-BY`, before the final shuffling (which has identical performance to sorting).

For an input relation with n rows, `DISTINCT` performs $n-1$ equality comparisons, one for each pair of adjacent rows. Since all these comparisons are independent, our vectorized implementation uses batching, thus, applying `DISTINCT` to the entire input in six rounds of communication (the number of rounds required for oblivious equality on pairs of 64-bit shares). As a result, `DISTINCT` scales well with the input size and can process $10M$ rows in $20s$. `GROUP BY` is slower than `DISTINCT`, as it requires significantly more rounds of communication, logarithmic to the input size. Finally, `ORDER BY` relies on our implementation of bitonic sort, where all $\frac{n}{2}$ comparisons at each level are batched within the same round.

Joins. The oblivious join operators in SECRECY hide the size of their output, thus, they compute the cartesian product between the two input relations and produce a bit share for all pairs of records, resulting in an output with $n \cdot m$ entries. We run both operators with $n = m$, for increasing input sizes, and plot the results in Figure 8b. The figure includes equi-join and range-join results for up to $100K$ rows per input, as we capped the duration of this experiment to $5h$. SECRECY executes joins in batches without materializing their entire output at once. As a result, it can perform $10B$ equality and inequality comparisons under MPC within the experiment duration limit.

D.3 EMP vs SECRECY on oblivious join

Here we compare EMP with SECRECY using an oblivious join operator that is based on the sample program from the SoK project [10]. For these experiments, we use inputs of the same cardinality and increase the size from $10K$ to $100K$ rows per input. We cap the time of these experiments to $12h$. Fig. 9 plots the results in `AWS-LAN` and `AWS-WAN`. Within the experiment duration, EMP can evaluate joins on up to $40K$ rows per input (in $11h$). SECRECY is $18\times$ faster for the same input size and can process up to $100K$ rows per input in less than $4h$.

E QUERIES USED IN THE PAPER

Here we list the queries used in §7 (in SQL syntax):

Comorbidity:
```
SELECT diag, COUNT(*) cnt
FROM diagnosis
WHERE pid IN cdiff_cohort
GROUP BY diag
ORDER BY cnt DESC
LIMIT 10
```

Recurrent C. Diff.:
```
WITH rcd AS (
   SELECT pid, time, row_no
   FROM diagnosis
   WHERE diag=cdiff)
 SELECT DISTINCT pid
 FROM rcd r1 JOIN rcd r2 ON r1.pid = r2.pid
 WHERE r2.time - r1.time >= 15 DAYS
 AND r2.time - r1.time <= 56 DAYS
 AND r2.row_no = r1.row_no + 1
```

Aspririn Count:
```
SELECT count(DISTINCT pid)
FROM diagnosis as d, medication as m on
d.pid = m.pid
WHERE d.diag = hd AND m.med = aspirin
AND d.time <= m.time
```

Password Reuse:
```
SELECT ID
FROM R
GROUP BY ID, PWD
HAVING COUNT(*)>1
```

Credit Score:
```
SELECT S.ID
FROM (
 SELECT ID, MIN(CS) as cs1, MAX(CS) as cs2
 FROM R
 WHERE R.year=2019
 GROUP-BY ID ) as S
WHERE S.cs2 - S.cs1 > c
```

TPC-H Q4:

```
SELECT o_orderpriority, count(*) as order_count
FROM orders
WHERE o_orderdate >= date '[DATE]' AND
 o_orderdate < date '[DATE]' + interval '3' month
 AND EXISTS (
      SELECT *
      FROM lineitem
      WHERE l_orderkey = o_orderkey
      AND l_commitdate < l_receiptdate
     )
GROUP BY o_orderpriority
ORDER BY o_orderpriority
```

TPC-H Q6:

```
SELECT sum(l_extendedprice*l_discount) as revenue
FROM lineitem
WHERE l_shipdate >= date '[DATE]' AND
 l_shipdate < date '[DATE]' + interval '1' year
 AND l_discount between [DISCOUNT] - 0.01
 AND [DISCOUNT] + 0.01 and l_quantity < [QUANTITY]
```

TPC-H Q13:

```
SELECT c_count, count(*) as custdist
FROM (
      SELECT c_custkey, count(o_orderkey)
      FROM customer left outer join orders ON
        c_custkey = o_custkey
        AND o_comment = ‘[WORD]’
      GROUP BY c_custkey
      ) as c_orders (c_custkey, c_count)
GROUP BY c_count
ORDER BY custdist desc, c_count desc
```