Secrecy: Secure collaborative analytics on secret-shared data

Secrecy: Secure collaborative analytics on secret-shared data

John Liagouris[†‡], Vasiliki Kalavri[†], Muhammad Faisal[†], Mayank Varia[†]
[†]Boston University, [‡]Hariri Institute for Computing
{liagos, vkalavri, mfaisal, varia}@bu.edu

ABSTRACT

We study the problem of composing and optimizing relational query plans under secure multi-party computation (MPC). MPC enables mutually distrusting parties to jointly compute arbitrary functions over private data, while preserving data privacy from each other and from external entities.

In this paper, we propose a relational MPC framework based on replicated secret sharing. We define a set of oblivious operators, explain the secure primitives they rely on, and provide an analysis of their costs in terms of operations and inter-party communication. We show how these operators can be composed to form *end-to-end oblivious queries*, and we introduce logical and physical optimizations that dramatically reduce the space and communication requirements during query execution, in some cases from quadratic to linear with respect to the cardinality of the input.

We provide an efficient implementation of our framework, called *Secrecy*, and evaluate it using real queries from several MPC application areas. Our results demonstrate that the optimizations we propose can result in up to 1000× lower execution times compared to baseline approaches, enabling *Secrecy* to outperform state-of-the-art frameworks and compute MPC queries on millions of input rows with a single thread per party.

1 INTRODUCTION

Cryptographically secure multi-party computation, or MPC for short, enables mutually distrusting parties to make queries of their collective data while keeping their own sensitive data siloed from each other and from external adversaries. Several MPC software libraries have been designed over the past decade that offer some combination of speed, scale, and programming flexibility (e.g., [1, 17, 20, 66, 81, 96]). MPC has been deployed to protect healthcare data like disease surveillance, educational data like student GPAs, financial data like credit modeling, advertising data like conversion rates, public interest data like the gender wage gap, and more [11, 16, 19, 29]. Nevertheless, adoption of MPC is rare, in part due to the challenge of developing and deploying MPC without domain-specific expertise [49].

To make secure computation more accessible to data analysts, systems like Conclave [88], ObliDB [36], OCQ [31], Opaque [98], SAQE [15], SDB [51, 92], Senate [79], Shrinkwrap [14], and SMCQL [13] are designed to compute relational queries while providing strong security guarantees. Despite their particular differences, these works aim to improve query performance either by sidesteping expensive MPC operations or by relaxing the full MPC security guarantees (or both).

We distinguish three main lines of work in this space: (i) works that rely on *trusted hardware* (e.g., secure enclaves [36, 98]) to avoid the inherent communication cost of MPC protocols, (ii) works that employ *hybrid execution* (e.g., [13, 88]) and split the query plan into a plaintext part (executed by the data owners) and an oblivious part (executed under MPC), and (iii) works that trade off secure query performance with *controlled information leakage*, e.g., by revealing information about intermediate result sizes to untrusted parties, either with noise [14, 15] or not [51, 88, 92]. More recently, Senate [79] combines hybrid execution in the spirit of SMCQL and Conclave with a technique that reduces joint computation under MPC by leveraging information about data ownership. Table 1 summarizes the features of the most prominent software solutions for relational analytics under MPC (we discuss hardware-based approaches in Section 7).

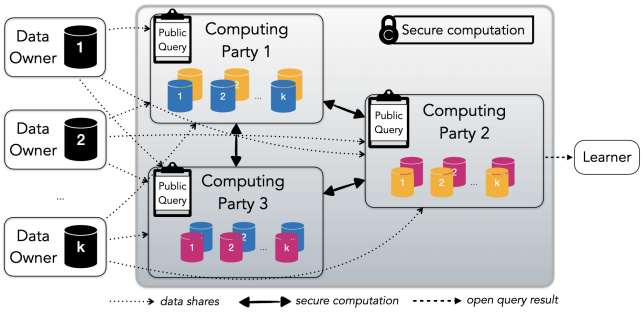


Figure 1: MPC setting overview for secure collaborative analytics

Although the frameworks listed in Table 1 propose various types of optimizations, these are applicable under certain conditions on data sensitivity, input ownership, and the role of data owners in the computation (cf. Optimization Conditions). For example, minimizing the use of secure computation via hybrid execution is only feasible when data owners can compute part of the query locally on their plaintext data (i.e. outside the MPC boundary). Moreover, SMCQL, SDB, and Conclave can sidestep MPC when attributes are annotated as non-sensitive, Shrinkwrap and SAQE calibrate leakage based on user-provided privacy budgets, and Senate reduces joint computation when some relations are owned by subsets of the computing parties.

In this paper, we study the fundamental problem of composing and optimizing MPC queries in a more challenging setting, where *all* data are sensitive and data owners *may not* have their own private resources to participate in the computation. In contrast to existing work that has sought to improve MPC query performance by either avoiding secure computation or relaxing its guarantees, we propose a set of optimizations for *end-to-end oblivious* queries that retain the *full* security guarantees of MPC. We contribute *Secrecy*, a framework for secure collaborative analytics that applies these optimizations, and we find that they can improve MPC query

Framework	MPC Protocol	Information Leakage	Trusted Party	Query Execution	Optimization Objective	Optimization Conditions
Conclave [88]	Secret Sharing / Garbled Circuits	Controlled (Hybrid operators)	Yes	Hybrid	Minimize the use of secure computation	1. Data owners participate in computation 2. Data owners provide privacy annotations 3. There exists a (fourth) trusted party
SMCQL [13]	Garbled Circuits / ORAM	No	No	Hybrid	Minimize the use of secure computation	1. Data owners participate in computation 2. Data owners provide privacy annotations
Shrinkwrap [14]	Garbled Circuits / ORAM	Controlled (Diff. Privacy)	No	Hybrid	Calibrate padding of intermediate results	1. Data owners participate in computation[1] 2. Data owners provide privacy annotations[1] and intermediate result sensitivities
SAQE [15]	Garbled Circuits	Controlled (Diff. Privacy)	No	Hybrid	Choose sampling rate for approximate answers	1. Data owners participate in computation[1] 2. Data owners provide privacy annotations[1] and privacy budget
Senate [79] [2]	Garbled Circuits	No	No	Hybrid	Reduce joint computation to subsets of parties	1. Data owners participate in computation 2. Input or intermediate relations are owned by subsets of the computing parties
SDB [51, 92] [3]	Secret Sharing	Yes (operator dependent)	No	Hybrid	Reduce data encryption and decryption costs	1. Data owner participates in computation 2. Data owner provides privacy annotations
Secrecy	Repl. Secret Sharing	No	No	End-to-end under MPC	Reduce MPC costs (Section 4)	None

[1] *Shrinkwrap and SAQE build on top of SMCQL's information flow analysis and inherit its optimizations along with their conditions.*
[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 1: Summary of MPC-based software solutions for relational analytics. Hybrid query execution is feasible when data owners participate in the computation. The rest of the optimizations supported by each system are applicable under one or more of the listed conditions.

performance by orders of magnitude. To the best of our knowledge, this is the first work to report results for oblivious queries on relations with up to millions of input rows, entirely under MPC, and without any information leakage or need for trusted hardware.

1.1 Framework overview

Figure 1 gives an overview of our MPC setting. A set of k data owners that wish to compute a public query on their private data distribute *secret shares* of the data to three untrusted computing parties. We adopt replicated secret sharing protocols (cf. Section 2.2), according to which each party receives two shares per input. The computing parties execute the query under MPC and open their results to a learner. Making such architectures for secure database services practical has been a long-standing challenge in the data management community [3, 4]. We design our MPC framework, *Secrecy*, on the following principles:

1. Decoupling data owners from computing parties. Contrary to existing works, *Secrecy* decouples the role of a computing party from that of a data owner. Our optimizations do not make any assumptions about data ownership and are all applicable even when none of the data owners participates in the computation.

2. No information leakage. *Secrecy* retains the full MPC security guarantees, that is, it reveals nothing about the data and the execution metadata to untrusted parties. It completely hides access patterns and intermediate result sizes.

3. No reliance on trusted execution environments. *Secrecy* does not rely on any (semi-)trusted party, honest broker or specialized secure hardware. To make our techniques accessible and remove barriers for adoption, we target general-purpose compute and cloud.

4. End-to-end MPC execution. *Secrecy* does not require data owners to annotate attributes as sensitive or non-sensitive and does not try to reduce the amount of secure computation. Instead, it executes all query operators under MPC and protects all attributes to prevent inference attacks that exploit correlations or functional dependencies in the data.

1.2 Contributions

We define a set of oblivious operators based on replicated secret sharing and describe how they can be composed to build complex MPC query plans. Our contributions are summarized as follows:

- We analyze the cost of oblivious operators and their composition with respect to the number of required operations, messages, and communication rounds under MPC.
- Based on this cost analysis, we propose a rich set of optimizations that significantly reduce the cost of oblivious queries: (i) database-style *logical transformations*, such as operator re-ordering and decomposition, (ii) *physical optimizations*, including operator fusion and message batching, and (iii) *secret-sharing optimizations* that leverage knowledge about the MPC protocol.
- We provide efficient implementations of the oblivious operators and corresponding optimizations in a new relational MPC framework called *Secrecy*.
- We evaluate *Secrecy*'s performance and the effectiveness of the proposed 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 will release *Secrecy* as open-source and make our experiments publicly available. This work aims to make MPC more accessible to the data management community and catalyze collaborations between cryptographers and database experts.

2 BACKGROUND ON MPC

Each party in MPC has one or more of the following roles:

- *Input party or data owner* that provides some input data.
- *Computing party*, e.g. a cloud provider that provides resources (machines) to perform the secure computation.
- *Result party or learner*, e.g. a data analyst who learns the output of the computation.

A party may have any combination of the above roles; in fact, it is quite common to have data owners acting as computing and/or result parties at the same time. This is also supported by *Secrecy* without affecting the security guarantees or the proposed optimizations. In addition, a party in MPC is a logical entity and does not necessarily correspond to a single compute node. For example, a cloud or IaaS provider can play the role of a single computing party that internally distributes its own part of the computation across a cluster of machines. *Secrecy* does not make any assumption about the parties' actual deployment, so it could be perfectly possible to deploy each party at competing providers or to have multiple providers in the same datacenter in a federated cloud.

Before using MPC, data owners must agree on the computation, in our setting a relational query, that they want to execute over the union of their private data. This query is *public*, i.e., known to all parties regardless of their role. To evaluate the query, computing parties execute an *identical* computation and exchange messages with each other.

2.1 Security guarantees and threat model

MPC broadly offers two types of security guarantees: *privacy*, meaning that nobody learns more than (what they can infer from) their own inputs and outputs, and *correctness*, meaning that the parties are convinced that the output of the calculation is accurate. These guarantees hold even in the presence of a dishonest *adversary* who controls a (strict) subset of the computing parties; different MPC protocols can withstand different adversary size and threat posture.

Most MPC protocols consider an adversary who corrupts an arbitrary threshold T of the N computing parties, although more complicated access control policies are possible. Also, most protocols consider an adversary who either passively attempts to break privacy while following the protocol (a "semi-honest" adversary) or one who is actively malicious and is therefore willing to deviate from the prescribed protocol arbitrarily. In this work, we focus on the setting of a *semi-honest* adversary, noting that there exist general transformations to the stronger malicious setting [37]. We discuss malicious security further in Section 8.

Concretely, the threat model of this work is as follows: we consider three computing parties, where the adversary has complete visibility into and control over the network through which these parties exchange messages. The adversary may add, drop, or modify packets at any time. Additionally, the adversary can passively monitor 1 of the 3 computing parties of their choice from the beginning of the protocol execution. Here, "passive monitoring" means that the adversary can view the contents of all messages received by this party and any data stored on the machine, but they cannot alter the execution of the corrupted party. We also assume that the software faithfully and securely implements the MPC protocol; that is, formal verification is out of scope for this work.

2.2 (Replicated) Secret Sharing

MPC protocols follow one of two general techniques: obscuring the truth table of each operation using Yao's *garbled circuits* [95], or interactively performing operations over encoded data using *secret sharing* [83]. Garbled circuits are an effective method to securely compute Boolean circuits in high-latency environments because they only need a few rounds of communication between computing parties. Secret sharing-based approaches require less overall bandwidth and support more data types and operators.

This work follows the approach of 3-party replicated secret sharing by Araki et al. [8]. We encode an ℓ-bit string of sensitive data s (*secret*) by splitting it into 3 shares s_1, s_2, and s_3 that individually have the uniform distribution over all possible n-bit strings (for privacy) and collectively suffice to specify s (for correctness). Next, we give each party P_i two of the shares s_i and s_{i+1}. Hence, any 2 parties can reconstruct a secret, but any single party cannot.

We consider two secret sharing formats: *boolean* secret sharing in which $s = s_1 \oplus s_2 \oplus s_3$, where $\oplus$ denotes the boolean XOR operation, and *additive* or *arithmetic* secret sharing in which $s = s_1 + s_2 + s_3 \mod 2^\ell$.

The computing parties are placed on a logical 'ring,' as shown in Figure 1. Given boolean secret sharings of two strings s and t or additive secret sharings of two values u and v, we describe next how the parties can collectively compute secret shares of many operations, without learning anything about the secrets.

2.3 Oblivious primitives

In this section, we briefly explain all primitives we use in our work.

Boolean operations. The parties can compute shares of $s \oplus t$ *locally*, i.e. without communication, by simply XORing their shares $s_i \oplus t_i$. To compute shares of the bitwise AND operation between s and t, denoted with $s \cdot t$ or simply st, one *round of communication* is required. Observe that $st = (s_1 \oplus s_2 \oplus s_3) \cdot (t_1 \oplus t_2 \oplus t_3)$. After distributing the AND over the XOR and doing some rearrangement we have $st = (s_1t_1 \oplus s_1t_2 \oplus s_2t_1) \oplus (s_2t_2 \oplus s_2t_3 \oplus s_3t_2) \oplus (s_3t_3 \oplus s_3t_1 \oplus s_1t_3)$. In our replicated secret sharing scheme, each party has two shares for s and two shares for t. More precisely, 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. Using its shares, each party can locally compute one of the three terms (in parentheses) of the last equation and this term corresponds to its boolean share of st. The parties then XOR this share with a fresh sharing of 0 (which is created locally [8]) so that the final share is uniformly distributed. In the end, each party must send the computed share to its successor on the ring (clockwise) so that all parties have two shares of st (without knowing the actual value of st) and the replicated secret sharing property is preserved. Logical OR and NOT operations are based on the XOR and AND primitives.

Equality/Inequality. The parties can collectively form a secret sharing of the bit b that equals 0 if and only if $s = t$ by first computing a sharing of $s \oplus t$ and then taking the boolean-AND of each of the bits of this string. Similarly, the parties can compare whether $s < t$ by checking equality of bits from left to right and taking the value of s_i at the first bit i in which the two strings differ.

By arranging the n fanin-2 AND gates in a log-depth tree, the number of communication rounds required for secure equality

(=, <>) and inequality (<, >, ≥, ≤) is $\lceil \log \ell \rceil$ and $\lceil \log(\ell + 1) \rceil$ respectively, where ℓ is the length of the operands in number of bits. For example, to check equality (resp. inequality) between two 64-bit integers, we need $\lceil \log 64 \rceil = 6$ (resp. $\lceil \log 65 \rceil = 7$) rounds. Note that it is possible to compute (in)equality in a constant number of rounds [30], but the constants are worse for typical string lengths.

Some special cases of (in)equality operators can be further optimized. Less-than-zero checks ($s < 0$) require a secret sharing of the most significant bit of s, which the parties already possess, so no communication is needed. Equality with a public constant $s \stackrel{?}{=} c$ can also be optimized by having the data owners compute two subtractions $s - c$ and $c - s$ locally (in the clear) and secret share the results. This way, checking $s = c$ is reduced in two oblivious inequalities $s - c < 0$ and $c - s < 0$, both of which are local. This optimization exists in other MPC frameworks as well [1], and we show later how we use it to evaluate selection locally.

Compare-and-swap. The parties can calculate the min and max of two strings. Setting $b = s \stackrel{?}{<} t$, 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 $\lceil \log(\ell + 1) \rceil$ rounds for the inequality plus two more rounds: one for exchanging shares of the computed bit b, and a second one to exchange the shares of the results of the four ANDs required by the multiplexer. Compare-and-swap overwrites the original strings s and t.

Sort and shuffle. A bitonic sorter, such as Batcher's sort [61], combines $O(n \log^2 n)$ compare-and-swap operators with a data-independent control flow. We can obliviously shuffle values in a similar fashion: each party appends a new attribute that is populated with locally generated random values, sorts the values on this new attribute, and then discards the new attribute (although we remark that faster oblivious shuffle algorithms are possible).

Boolean addition. In case s and t are integers, computing the share of $s + t$ can be done in ℓ rounds of communication using a ripple-carry adder [59]. Rounds can be further reduced to $O(\log \ell)$ with a parallel prefix adder, at the cost of exchanging more data.

Arithmetic operations. Addition using additive shares is more efficient. Given additive shares of two secrets u and v, parties can compute $u + v$ locally. Multiplication $u \cdot v$ is equivalent to a logical AND using boolean shares, so it requires one round of communication as explained above. Scalar multiplication is local.

Conversion. We can convert between additive and boolean sharings [68] by securely computing all of the XOR and AND gates in a ripple-carry adder. One special case of conversion that is useful in many cases is the boolean-to-arithmetic conversion of shares for single-bit secrets. This conversion can be done in two rounds with the simple protocol used in [1]. We explain how we leverage this optimization to speedup oblivious aggregations later.

3 OBLIVIOUS RELATIONAL OPERATORS

In this section, we define the oblivious operators of *Secrecy*, analyze their cost, and describe how they can be composed. 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 oblivious sorting. In all cases, the operator's predicate is evaluated under MPC using the primitives of Section 2.3.

Our oblivious operators hide both *access patterns* and *output size* from the computing parties. We hide access patterns by implementing the operator in a way that makes its control-flow independent of the input data so that it incurs exactly the same accesses for all inputs of the same size. In practice, this means that the implementation does not include any `if` statements that depend either directly or indirectly on the input data. Also, all operators except `PROJECT` and `ORDER-BY` introduce a new single-bit attribute that stores the (secret-shared) result of a logical or arithmetic expression evaluated under MPC. This extra attribute denotes whether the respective tuple belongs to the output of an oblivious operator and is always discarded before opening the final result to the learner(s). Along with 'masking' that we describe below, the single-bit attribute enables the computing parties to jointly apply each operator without learning the actual size of any intermediate or output relation.

3.1 Individual 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 how each operator is computed over the logical (i.e. secret) relations and not the actual shares distributed across parties. That is, when we say that "a computation is applied to a relation R and defines another relation T", in practice this means that each computing party begins with shares of R, performs some MPC operations, and ends with shares of T.

PROJECT. Oblivious projection has the same semantics as the non-oblivious operation.

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 attribute for each tuple $t \in R$ that contains ϕ's result when applied to t (each party has two shares of the actual result according to the replicated secret sharing protocol). The result is a single bit denoting whether the tuple t is included in T (1) or not (0). The predicate ϕ can be an arbitrary logical expression with atoms that may also include arithmetic expressions $(+, *, =, >, <, \neq, \geq, \leq)$. Such expressions are evaluated under MPC using the primitives of Section 2.3. Note that, in contrast to a typical selection in the clear, oblivious selection defines a relation with the same cardinality as the input relation, i.e., it does not remove tuples from the input so that the size of the output remains hidden to the computing parties.

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'). This result is a cartesian product of the input relations ($R \times S$), where each tuple is augmented with a single bit (0/1) denoting whether the tuple t "matches" with tuple t' according to θ. Generating the cartesian product is inherent to general oblivious join algorithms (we discuss special join instances

in Section 7). Like selections, the join predicate can be an arbitrary logical expression with atoms that may also include arithmetic expressions. Join is the only oblivious operator in *Secrecy* that generates a relation with cardinality larger than the cardinalities of its inputs.

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 $\bigvee_{\forall t' \in S} \theta(t, t')$ indicating whether the row in R "matches" with any row in S.

ORDER-BY. Oblivious order-by on attribute a_k has the same semantics as the non-oblivious operator, where each tuple is assigned an index i such that:

$$\forall t_i, t_j \in R, i < j \iff \begin{cases} t_i[a_k] < t_j[a_k] \text{ (ASC)} \\ \\ t_i[a_k] > t_j[a_k] \text{ (DESC)} \end{cases}$$

The tuple ordering is computed under MPC using oblivious compare-and-swap operations (cf. Section 2.3). 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. Limit in the oblivious setting has the same semantics. Order-by with limit is the only operator in *Secrecy* that may output a relation with cardinality smaller than the cardinality of its input.

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., $t_1[a_w] = t_2[a_w] = ... = t_u[a_w], 1 \leq u \leq |R|$. The function f in T's definition above is defined as:

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

where t_{inv} is a tuple with $t_{inv}[a_v] = 0$ and the rest of the attributes set to a special invalid value, while $agg(S)$ is the aggregation function, e.g. `MIN`, `MAX`, `COUNT`, `SUM`, `AVG`. 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." Groups can be defined on multiple attributes (keys) using the standard semantics. Global oblivious aggregation on attributes of R is defined by assigning all tuples in R to a single group.

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:

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

In simple words, distinct marks one tuple per 'group' as 'valid' and the rest as 'invalid'.

MASK. Let t_{inv} be a special tuple with invalid attribute values. 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.

3.2 Cost of oblivious operators

We now describe the implementation of oblivious operators and analyze their individual costs before discussing plan composition in the next section. In *Secrecy*, we have chosen to provide general implementations that favor composability. Building a full-fledged MPC planner that considers alternative operator implementations and their costs is out of the scope of this paper but certainly an exciting opportunity for follow-up work (cf. Section 8).

We consider two types of costs for individual operators: (i) *operation costs* defined in terms of the total *number of MPC operations* per party, which include local computation and message exchange, and (ii) *synchronization costs* for inter-party communication, which we measure by the number of communication rounds across parties. All secret-shared data in our framework reside in main memory, therefore, we do not consider disk I/O costs.

A communication round corresponds to a single clockwise data exchange on the ring between the 3 computing parties. In practice, this is a *barrier*, i.e. a synchronization point in the distributed computation, where parties must exchange data in order to proceed. In general, the fewer rounds an operation needs the faster it reaches completion since each party can make more progress without being blocked on other parties. Table 2 shows the number of operations as well as the communication rounds required by each individual operator with respect to the input size. Throughout this section, we use n, m to refer to the cardinalities of input relations and ℓ to denote the length (in bits) of a secret-shared value.

PROJECT. The cost of an oblivious `PROJECT` is the same as its plaintext counterpart: it does not require any communication, as each party can locally disregard the shares corresponding to the filtered attributes.

SELECT. In terms of operations, oblivious `SELECT` performs a linear scan of the input relation R. Because predicate evaluation can be computed independently for an arbitrary number of rows, the number of rounds (i.e., synchronization barriers) to perform the `SELECT` equals the number of rounds required to evaluate the selection predicate on a *single* row; it is independent of the size of R.

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	$O(n \cdot \log^2 n)$	$O(n)$
MASK	$O(n)$	$O(1)$

Table 2: Summary of operation and synchronization costs 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. These messages can be batched in rounds as shown in the rightmost column. JOIN is the most expensive operator in number of operations/messages whereas GROUP-BY is the most expensive operator in number of rounds.

In Section 4.4, we describe a technique we use in *Secrecy* that can reduce selections to local operations.

JOIN. Oblivious JOIN is the most expensive operation in terms of operation cost as it requires a nested-loop over the input relations to check all possible pairs ($n \cdot m$); however, the number of communication rounds in the oblivious JOIN is independent of the input sizes n and m. As in the case of SELECT, the number of rounds only depends on the join predicate. For equality joins, each one of the $n \cdot m$ equality checks requires $\lceil \log \ell \rceil$ rounds (where ℓ is the length of the join attributes in bits) and is independent of others, hence, the whole join can be done in $\lceil \log \ell \rceil$ rounds. Range joins are more expensive. A range join with predicate of the form $R.a \leq S.b$, where a, b are attributes of the input relations R and S, requires $\lceil \log(\ell + 1) \rceil$ rounds in total. The constant asymptotic complexity with respect to the input size holds for *any* θ-join.

SEMI-JOIN. Oblivious semi-joins require the same number of operations as the θ-joins but the number of communication rounds is different. A semi-join $R \ltimes_\theta S$ requires $O(\log |S|)$ communication rounds to evaluate the formula $\bigvee_{\forall t' \in S} \theta(t, t')\}$ from Section 3.1. This formula requires ORing $|S|$ bits, which can be done in $\lceil \log |S| \rceil$ communication rounds by using a binary tree of logical operators, as in the case of equality and inequality (cf. Section 2.3).

ORDER-BY. Oblivious ORDER-BY relies on Bitonic sort that performs $O(n \cdot \log^2 n)$ compare-and-swap operations in $\frac{\log n(1+\log n)}{2}$ stages, where each stage involves $\frac{n}{2}$ independent compare-and-swap operations that can be performed *in bulk*. In this case, the number of messages required by each oblivious compare-and-swap is linear to the number of attributes in the input relation, however, the number of rounds depends only on the cardinality of the input. Given the number of rounds of each compare-and-swap operation (cf. Section 2.3), the total number of rounds required by ORDER-BY is:

$$\log n \cdot (1 + \log n) \cdot (1 + 1/2 \cdot \lceil \log(\ell + 1) \rceil)$$

where n is the cardinality of the input relation, and ℓ is the length of the sort attribute in bits. The analysis assumes one sorting attribute. Adding more sorting attributes increases the number of rounds in each comparison by a small constant factor.

```
1 sort input relation R on a_k;
2 for each pair of adjacent tuples (t_i, t_{i+1}), 0 ≤ i < |R|, do
      //Are tuples in the same group?
3     let b ← t_i[a_k] ≟ t_{i+1}[a_k];
      //Aggregation
4     t_{i+1}[a_g] ← b · agg(t_i[a_w], t_{i+1}[a_w]) + (1 − b) · t_{i+1}[a_g];
5     t_i[a_v] ← ¬b;
      //Masking
6     for each attribute a ≠ a_v of t_i do
7         let r be an invalid value;
8         a ← b · r + (1 − b) · a;
9 shuffle R;
```

Algorithm 1: Main control-flow of oblivious group-by

GROUP-BY. The logic of oblivious group-by is given in Algorithm 1. Let a_k be the group-by key, a_w the aggregated attribute, a_g the extra attribute that stores the aggregation result, and a_v the 'valid' bit (same notation as in Section 3.1). The first step is to sort the input relation on the group-by key (**line 1**). Then, the operator scans the sorted relation and, for each pair of adjacent tuples, applies an oblivious equality comparison on a_k (**line 3**). The result of this comparison (b) is used to aggregate (**line 4**), set the 'valid' bit (**line 5**), and "mask" (**lines 6-8**) obliviously. Aggregation is updated incrementally based on the values of the last pair of tuples (**line 4**). MIN, MAX, COUNT, and SUM can be easily evaluated this way but for AVG we need to keep the sum (numerator) and count (denominator) separate. When the scan is over, the algorithm requires a final shuffling (**line 9**) to hide the group "traces" in case the relation (or a part of it) is opened to the learner; this step is only needed if no subsequent sorting is required in the query plan, which would obliviously re-order R anyway.

This operator is the most expensive in terms of communication rounds because the aggregation function is applied *sequentially* on each pair of adjacent tuples. Accounting for the initial sorting and final shuffling, the total number of rounds required by GROUP-BY is:

$$(n - 1) \cdot c_{agg} + \log n \cdot (1 + \log n) \cdot (2 + \lceil \log(\ell + 1) \rceil)$$

where c_{agg} is the number of rounds required to apply the aggregation function to a pair of rows (independent of n).

Aggregations. Aggregations can be used without a GROUP-BY clause. In this case, applying the aggregation function requires $n - 1$ operations in total but the number of communication rounds can be reduced to $O(\log n)$ by building a binary tree of function evaluations. This optimization makes aggregations efficient in practice, and other works have used it to reduce the number of rounds in GROUP-BY if the data owners agree to reveal the group sizes [22, 55].

DISTINCT. Distinct is a special case of group-by where a_k is in this case the distinct attribute. As such, it follows a slightly different version of Algorithm 1 where, for each pair of adjacent tuples, we apply the equality comparison on a_k (**line 3**) and set the distinct bit $t_{i+1}[a_v]$ to $\neg b$ (the value $t_0[a_v]$ of the first tuple is set to 1). Lines **4-9** are simply omitted in this case because distinct does not require aggregation, masking or shuffling. Crucially, each evaluation of the loop is independent, so the communication rounds of the equality comparisons (**line 3**) can be performed *in bulk* for all pairs of tuples.

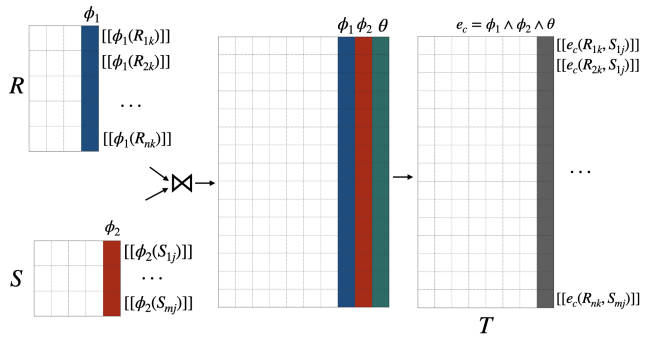


Figure 2: Example composition of two oblivious selections and a θ-join. The single-bit attributes ϕ_1, ϕ_2, and θ are used to compute the final attribute e_c that denotes whether a row belongs to the result T and is secret-shared amongst computing parties. The cost of the composition in this case is the cost of evaluating the logical expression $e_c = \phi_1 \wedge \phi_2 \wedge \theta$ under MPC, for each tuple in T. All $|T|$ expressions are independent and can be evaluated in bulk within two communication rounds, one for each logical AND ($\wedge$) in e_c's formula.

Hence, oblivious DISTINCT requires the same asymptotic number of operations as ORDER-BY because its operation cost is dominated by the initial sort (**line 1**). DISTINCT's communication cost is also dominated by that of ORDER-BY; the only extra effort is to compute $n-1$ equality checks in bulk, yielding the following total cost:

$$\log n \cdot (1 + \log n) \cdot (1 + 1/2 \cdot \lceil \log(\ell + 1) \rceil) + \lceil \log \ell \rceil$$

MASK. The cost of MASK is similar to the cost of SELECT; it requires n operations and a constant number of communication rounds to apply the masking function.

3.3 Composing oblivious operators

Consider the composition of two operators defined as applying the second operator to the output of the first operator. One merit of our approach is that all operators of Section 3.1 reveal nothing about their output or access patterns, so they can be arbitrarily composed into an *end-to-end oblivious query* plan without special treatment.

Let op_1 and op_2 be two oblivious 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) because 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 tuples generated by op_2. The expression e_c depends on the types of operators, as described below. Table 3 summarizes the composition costs for different operator pairs in *Secrecy*.

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 $|T|$ boolean ANDs all of which are independent and can be evaluated in one round. An example of composing two oblivious selections with an oblivious θ-join is given in Figure 2.

Composing distinct with other operators. Applying a selection or a (semi-)join to the result of DISTINCT requires a single communication round in order to compute the conjunction of the selection or (semi-)join bit with the bit a_υ generated by distinct. However, applying DISTINCT to a relation derived by 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_υ at each tuple, we need to make sure there are no other tuples with the same attribute a_k, with $a_\phi = 1$, and whose distinct bit a_υ is already set. More formally:

$$t_i[a_\upsilon] = \begin{cases} 1, \text{ iff } \nexists t_j, i \neq j : t_i[a_k] = t_j[a_k] \wedge t_j[a_\phi] = 1 \wedge t_j[a_\upsilon] = 1 \\ 0, \text{ otherwise} \end{cases}$$

To evaluate the above formula, 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 constant number of rounds required by distinct when applied to a base relation (cf. Table 2). Applying distinct to the output of a group-by or (semi-)join incurs a linear number of rounds for the same reason. In Section 4.3, we propose 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:

$$b \leftarrow b \wedge t_i[a_\phi] \wedge t_{i+1}[a_\phi]$$

at each step of the for-loop in Algorithm 1, where b is the bit that denotes whether the tuples t_i and t_{i+1} belong to the same group (**line 3** in Algorithm 1) and a_ϕ is the selection bit. This formula includes two logical 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 logical AND between the selection bit a_ϕ and the 'valid' bit a_υ 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 logical 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.

Operator pair(s)	#comm. rounds
{SELECT, (SEMI-)JOIN} → DISTINCT	$O(n)$
DISTINCT → {SELECT, (SEMI-)JOIN}	$O(1)$
SELECT ↔ (SEMI-)JOIN	$O(1)$
GROUP-BY → {SELECT, (SEMI-)JOIN}	$O(1)$
{SELECT, (SEMI-)JOIN} → GROUP-BY	$O(n)$
{GROUP-BY, DISTINCT} ↔ {GROUP-BY, DISTINCT}	$O(n)$

Table 3: Summary of composition costs in number of rounds for pairs of oblivious operators in *Secrecy w.r.t* the number of generated tuples (n). Arrows denote the order of applying the two operators. Composition incurs a small constant number of boolean operations per tuple, so the cost in number of operations is $O(n)$ for all pairs.

4 OPTIMIZATIONS FOR OBLIVIOUS QUERIES

In this section, we present the set of optimizations in *Secrecy*: logical transformation rules, such as operator reordering and decomposition (Section 4.2), physical optimizations, such as message batching and operator fusion (Section 4.3), and secret-sharing optimizations that further reduce the number of communication rounds for certain operators (Section 4.4). Finally, in Section 4.5, we show concrete examples of the cost reduction our optimizations achieve when applied on real-world queries.

Target queries. In this work, we focus on collaborative analytics under MPC where two or more data owners want to make queries on their collective data without compromising privacy. We consider all query 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 collaborative MPC to compute a query that includes a patient's name along with their diagnosis in the SELECT clause would be pointless. Thus, we target queries that return global or per-group aggregates and/or distinct results.

4.1 Optimization rationale

Cost-based query optimization on plaintext data relies on selectivity estimation to reduce the size of intermediate results. The oblivious operators in *Secrecy*, however, hide the true size of their results by producing fixed-size outputs for all inputs of the same cardinality. As a consequence, traditional cost-based optimization techniques for relational queries are not always effective when optimizing plans under MPC. Consider, for instance, the case of the ubiquitous "filter push-down" transformation rule. Since oblivious selections do not reduce the size of intermediate data, this transformation does not improve the cost of operators following the filter.

To define optimizations that are effective under MPC, we instead aim to minimize the cost of oblivious queries. The total cost of a query plan can be computed as a function of the individual costs provided in Tables 2 and 3. In particular:

- The **operation cost**, which is determined by the total number of operations and messages per party (Section 3.2).
- The **synchronization cost**, given by the number of communication rounds across parties (Section 3.2).
- The **cost of composition**, which is also measured in number of operations and communication rounds (Section 3.3).

Observations. The optimization rules we present in this section are guided by the following observations:

(1) With the exception of LIMIT, oblivious operators never reduce the size of intermediate data.
(2) JOIN is the only operator that produces an output larger than its input.
(3) The synchronization cost of the blocking operators, ORDER-BY, GROUP-BY, and DISTINCT, depends on the size of their input.
(4) When DISTINCT follows a selection, a (semi-)join or a group-by, the total asymptotic cost of composition increases from a constant to a linear number of rounds *w.r.t.* the input size.

4.2 Logical transformation rules

Guided by observations (1)-(3), we propose three logical transformation rules that reorder and decompose pairs of operators to lower the cost of oblivious query plans. Although non-standard, the rules we describe in this section are valid algebraic transformations for plaintext queries and there are no special applicability conditions in the secure setting.

4.2.1 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 reduces all three cost factors and can provide significant performance improvements in practice, even if the asymptotic costs do not change. As an example, consider the case of applying ORDER-BY before a selection. Recall that the number of operations and messages required by the oblivious ORDER-BY depends on the cardinality and the number of attributes of the input relation (cf. Section 3.2). Applying the selection after the order-by reduces the actual (but not the asymptotic) operation cost, as selection appends one attribute to its input.

4.2.2 Join push-up. The second transformation rule is guided by observation (2) 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 operators whose synchronization cost depends on the input size. For example, placing a blocking operator after a join requires sorting the cartesian product of the input relations, which increases the synchronization cost of a subsequent GROUP-BY to $O(n^2)$ and the operation cost of any following blocking operator to $O(n^2 \log^2 n)$.

Similar re-orderings have been proposed for plaintext queries [26, 94], however, in the MPC setting this transformation does not reduce the size of intermediate data. Note that, under MPC, a plan that applies ORDER-BY on a JOIN input produces exactly the same amount of intermediate data as a plan where ORDER-BY is placed after JOIN, yet the latter plan has a higher cost.

Example. Consider the following query:

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

```
sort input relation R on a_θ, a_k;
for each pair of adjacent tuples (t_i, t_{i+1}), 0 ≤ i < |R|, do
    //Are tuples in the same group?
    let b ← t_i[a_k] ?= t_{i+1}[a_k];
    //Are tuples in the semi-join output too?
    let b_c ← b ∧ t_i[a_θ] ∧ t_{i+1}[a_θ];      //b_c is a single bit
    //Aggregation
    t_{i+1}[a_g] ← b_c · (t_i[a_g] + t_{i+1}[a_g]) + (1 − b_c) · t_{i+1}[a_g];
    t_i[a_v] ← ¬b_c;                            //a_v is the 'valid' bit
    //Masking
    for each attribute a ≠ a_v of t_i do
        let r a random value;
        a ← b_c · r + (1 − b_c) · a;
shuffle R;
```

Algorithm 2: Second phase of the Join-Aggregation decomposition.

Let R and S have the same cardinality n. A plan that applies DISTINCT after the join operator requires $O(n^2 log^2 n)$ operations and messages per party. On the other hand, pushing DISTINCT before JOIN reduces the operation cost to $O(n \log^2 n)$ and the composition cost from $O(n^2)$ to $O(n)$ (in number of operations) and $O(1)$ (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.

4.2.3 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 produce 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 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 (IN) on a_j, which appends two attributes to R: an attribute a_θ introduced by the semi-join, and a second attribute a_g introduced by the group-by (cf. Section 3.1)[1]. During this step, a_g is initialized with a partial aggregation for each tuple in R. The partial aggregation depends on the aggregation function in the query (we provide an example below).

In the second phase, we compute the final aggregates per a_k using Algorithm 2, which takes into account the attribute a_θ and updates the partial aggregates a_g in-place with a single scan over R. The decomposition essentially replaces the join with an equivalent 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 plaintext (at the data owners) and then computing the global aggregates under MPC [13, 79]; in our case, all data are secret-shared amongst parties and *both* phases are under MPC.

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

The decomposition rule works for all common SQL aggregations (SUM, COUNT, MIN/MAX, AVG). It can also be used to push down DISTINCT in queries like **Q1** when the distinct attribute is different from the join attribute. In this case, there is no partial aggregation; we simply do the semi-join that appends the attribute a_θ (as above) and, in the second phase, we apply the distinct operator to R by taking into account a_θ.

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 $O(n^2)$ communication rounds. When decomposing the aggregation in two phases, the operation cost is reduced to $O(n \log^2 n)$ (due to oblivious sorting of R) and the synchronization cost is reduced to $O(n)$ rounds (due to the final grouping on R). The space requirements are also reduced from $O(n^2)$ to $O(n)$. In this example, the partial aggregation amounts to summing (under MPC) the $|S|$ bits produced by the semi-join in the first phase of the decomposition.

4.3 Physical optimizations

In this section, we describe a set of physical optimizations in *Secrecy* that further reduce the cost of oblivious plans.

4.3.1 Predicate fusion. Fusion is a common optimization in plaintext query planning, where the predicates of multiple filters can be merged and executed by a single operator. Fusion is also applicable to oblivious selections and joins with equality predicates, and is essentially reduced to identifying independent operations that can be executed 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 $2\lceil \log \ell \rceil + 1$. Next, we describe a somewhat more interesting fusion.

4.3.2 Distinct fusion. Recall that applying DISTINCT after SELECT requires n communication rounds (cf. Section 4.1, Observation (4)). 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 equality or inequality predicate of the (semi-)join.

4.3.3 Message batching. In communication-intensive MPC tasks, each non-local operation requires exchanging a constant number of messages, which in practice are very small in size (i.e., a few bytes). Grouping and exchanging small independent messages in bulk improves performance significantly. 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 (cf. Section 2.3). Applying the selection tuple-by-tuple and sending messages eagerly (as soon as they are generated) results in $n \cdot \lceil \log \ell \rceil$ communication 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 apply this optimization by default to all oblivious operators in *Secrecy*. Costs in Tables 2 and 3 already take message batching into account.

4.4 Secret-sharing optimizations

Secrecy uses boolean sharing by default, however, computing arithmetic expressions or aggregations, e.g. `COUNT` and `SUM`, on boolean shares requires using a ripple-carry adder, which in turn requires inter-party communication. On the other hand, the same operations on additive shares are local to each computing party. In this section, we describe two optimizations that avoid the ripple-carry adder in aggregations and predicates with constants.

4.4.1 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 the ripple-carry adder (cf. Section 2.3), 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 Section 4.2.3 that applies an aggregation function to the output of a join according to Algorithm 2. The attribute a_θ in Algorithm 2 is a single-bit attribute denoting that the respective row 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 **4**. The naïve approach is to evaluate the following equivalent logical expression directly on the boolean shares of b_c, $t_i[a_g]$, and $t_{i+1}[a_g]$:

$$t_{i+1}[a_g] \leftarrow b_\ell \wedge \mathsf{RCA}\Big(t_i[a_g], t_{i+1}[a_g]\Big) \oplus \overline{b}_\ell \wedge t_{i+1}[a_g]$$

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 **4** of Algorithm 2 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 **4**, they exchange boolean as well as arithmetic shares of its value. To do this efficiently, we rely on the single-bit conversion protocol used also in CrypTen [1], which only requires two rounds of communication. Having boolean and arithmetic shares of b_c allows us to use it in boolean and arithmetic expressions without paying the cost of `RCA`.

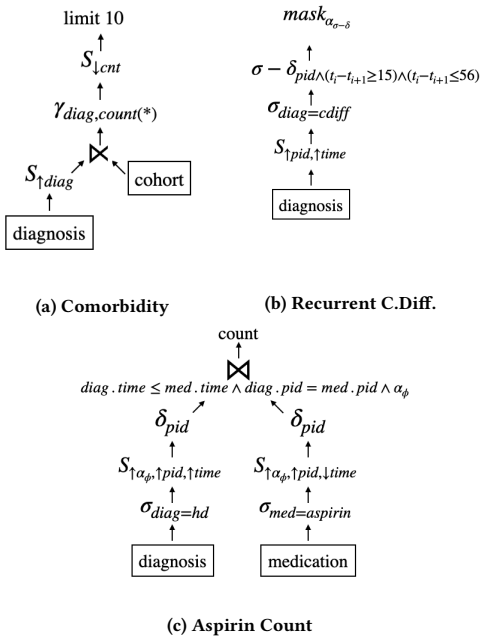


Figure 3: Optimized query plans for three real queries

4.4.2 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 the ripple-carry adder. 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 (cf. Section 2.3). 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 not padded). In our case, parties simply receive additional shares and will not learn anything about the intermediate query results.

4.5 Optimizations on real queries

We now showcase the applicability of *Secrecy*'s optimizations on three queries from clinical studies [52, 74, 84] that have also been used in other MPC works [13–15, 79, 88]. We experimentally evaluate the performance benefits on a larger set of queries in Section 6.

Comorbidity. This query returns the ten most common diagnoses of individuals in a cohort.

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

This query lends itself to join-aggregation decomposition and dual sharing, producing the plan shown in Figure 3a. Let n be the cardinality of `diagnosis`. The number of operations needed to evaluate this query is $O(n \log^2 n)$ (due to oblivious sorting) whereas the number of communication rounds is $O(n)$ (due to the oblivious group-by). The space requirements are bounded by the size of `diagnosis`, i.e., $O(n)$.

Recurrent Clostridium Difficile. This query returns the distinct ids of patients who have been diagnosed with `cdiff` and have two consecutive infections between 15 and 56 days apart.

```
WITH rcd AS (
    SELECT pid, time, row_no() OVER
    (PARTITION BY pid ORDER BY time)
    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
```

Two optimizations are applicable in this case. First, we apply blocking operator push-down to sort on `diagnosis` before applying the selection. Second, we use distinct fusion (σ-δ) to evaluate the inequality predicates along with `DISTINCT`. The optimized plan is shown in Figure 3b and it requires $O(n \log^2 n)$ operations and $O(\log^2 n)$ communication rounds. Note that an end-to-end oblivious implementation of the plan used in [13] requires $O(n^2 \log^2 n)$ operations and 4× more communication rounds, i.e., $O(\log^2 n^2) = O(4 \log^2 n) = O(\log^2 n)$. This is because `PARTITION BY` is not possible under MPC without revealing the number of partitions and, thus, the self-join will generate and materialize the cartesian product `rcd` × `rcd`, before applying the final `DISTINCT` operation.

Aspirin Count. The third query returns the number of patients who have been diagnosed with heart disease and have been prescribed `aspirin` after the diagnosis was made.

```
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
```

Here, we use blocking operator push-down and join push-up. We push the blocking distinct operator after the join to avoid materializing and sorting the join output. The optimized plan is shown in Figure 3c. Let `diagnosis` and `medication` have the same cardinality n. The number of operations needed to evaluate the query is $O(n^2)$ whereas the number of communication rounds is $O(\log^2 n)$. In contrast, an end-to-end oblivious implementation of the plan in [13] requires $O(n^2 \log^2 n)$ operations and 4× more rounds, since it applies distinct to the materialized join output.

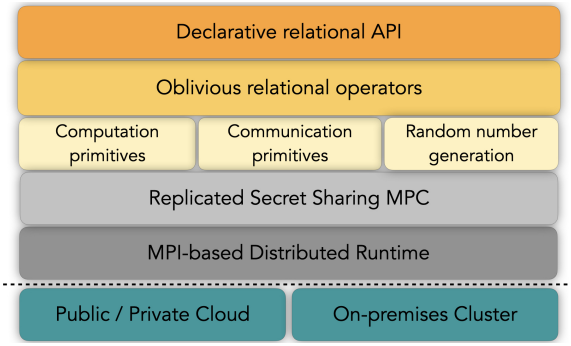


Figure 4: Overview of the *Secrecy* architecture.

```
/** Commorbidity Query **/
BTable t1 = get_shares(diagnosis);
BTable t2 = get_shares(cohort);

// Sort t1 on diag (at index 2)
bitonic_sort(&t1, 2, ASC);
in(&t1, &t2, 0, 0); // Semi-join on pid
group_by_count(&t1, 2); // Group-by on diag
// Sort t1 on count (at index 4)
bitonic_sort(&t1, 4, ASC);
open(t1, 10); // Open first 10 rows
```

Figure 5: Comorbidity query with *Secrecy*'s API.

5 *SECRECY* IMPLEMENTATION

Even though there exist various open-source MPC frameworks [49], we decided to implement *Secrecy* entirely from scratch. As a result, we were able to design and implement secure low-level primitives and oblivious operators that are optimized to process shares of *tables* instead of single attributes. In this section, we provide a brief description of the most important *Secrecy* implementation aspects.

Architecture overview. Figure 4 shows an overview of the *Secrecy* framework. *Secrecy* is implemented in C and can be deployed on local clusters or machines in the cloud. The distributed runtime and communication layer are based on MPI[2]. *Secrecy* currently does not encrypt data in transit between parties but it can be easily combined with any TLS implementation or other networking library that does so. Each computing party is a separate MPI process and we currently use a single thread per party to handle both local computation and communication with other parties. Parties are logically placed on a *ring* as shown in Figure 1. The middle layers of *Secrecy* include our implementation of the replicated secret sharing protocol, a library of secure computation and communication primitives, and the random number generation protocols. We built the latter with the `libsodium` library[3]. The upper two layers of the stack provide optimized implementations of the oblivious relational operators and a declarative relational API.

Query execution. Upon startup, the parties establish connections to each other and learn the process ids of their successor and predecessor parties. Then, they construct a *random sharing of zero* generator, so that they can jointly create random shares of the value 0 for the various secure primitives. To achieve that at scale, parties generate a random seed and share it with their successor in the ring. This way, each party has access to one local and

[2]https://www.mcs.anl.gov/research/projects/mpi/standard.html
[3]https://libsodium.gitbook.io

one remote pseudo-random number generator: $rand1$ and $rand2$. The parties can now generate a random share of 0 on demand as $sz = rand1.get\_next() - rand2.get\_next()$. Next, they receive input shares for each base relation from the data owners.

Queries are specified in a declarative API that allows composing operators seamlessly and abstracts the communication and MPC details. To compute the result of a query, parties execute an identical piece of code on their data shares. As an example, Figure 5 shows the *Secrecy* code that implements the Comorbidity query from Section 4.5 (we omit two function calls that convert boolean shares to arithmetic for brevity). We use a `64-bit` data representation for shares, so in our implementation $\ell = 64$ (cf. Section 3).

Configurable batching. Primitives and relational operators in *Secrecy* operate in *batched* mode, that is, they provide the ability to process multiple table rows in bulk and batch independent messages into a single round of communication (cf. Section 4.3.3). 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 without running out of memory or switching to an expensive disk-based evaluation.

6 EXPERIMENTAL EVALUATION

Our experimental evaluation is structured into five parts:

Performance on real and synthetic queries. In Section 6.2, we evaluate *Secrecy* on eight real and synthetic queries. We show that *Secrecy*'s implementation is efficient and its optimizations effectively reduce the runtime of complex queries by up to three orders of magnitude. In contrast to the baseline plans that fail to scale for inputs beyond a few thousand records, *Secrecy* can process hundreds of thousands and up to millions of input rows, entirely under MPC, in reasonable time.

Comparison with state-of-the-art frameworks. In Section 6.3, we compare *Secrecy* with two state-of-the-art MPC frameworks: SMCQL [13] and EMP [91]. We show that *Secrecy* outperforms both of them and can comfortably process much larger datasets within the same amount of time.

Benefits of optimizations. In Section 6.4, we evaluate the benefits of *Secrecy*'s logical, physical, and secret-sharing optimizations on the three queries of Sections 4.2-4.4. 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 *Secrecy*'s dual sharing improve execution time by 2×.

Performance of relational operators. In Section 6.5, we present performance results for individual relational operators. We show that *Secrecy*'s batched operator implementations are efficient and that by properly adjusting the batch size, they can comfortably scale to millions of input rows without running out of memory.

Micro-benchmarks. Finally, in Section 6.6, we drill down and evaluate individual secure computations and communication primitives that relational operators rely upon. We empirically verify the theoretical cost analysis of Section 2.3, evaluate the scalability of primitives, and quantify the positive effect that message batching has on the performance of communication-heavy operations.

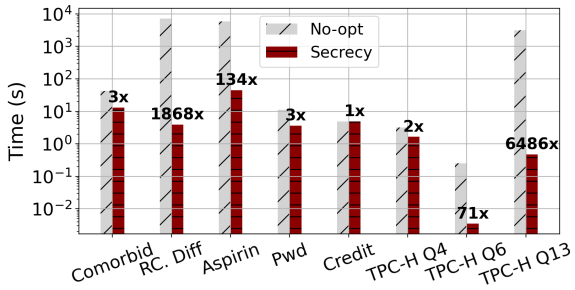


Figure 6: Performance gains of *Secrecy*'s optimizations over baseline plans for real and synthetic queries. Logical and physical optimizations result in over 100× lower execution times, while secret-sharing optimizations improve query performance by up to 71×.

6.1 Evaluation setup

We run all experiments on a three-node cluster of VMs in the Massachusetts Open Cloud (MOC) [2]. Each VM has 32GB of memory and 16 vCPUs and runs Ubuntu `16.04.12`, `C99`, `gcc 5.4.0`, and `MPICH 1.4`. Each MPC party is assigned to a different VM and runs as a single MPI process. For the purpose of our experiments, we designate one party as the data owner that distributes shares and reveals results in the end of the computation. Reported measurements are averaged over at least three runs and are plotted in log-scale, unless otherwise specified.

Queries. We use 11 queries in total. Five of them are real-world queries that have also been used in previous MPC works [13–15, 79, 88]. We use the three medical queries from [13] (*Comorbidity*, *Recurrent C.Diff.*, and *Aspirin Count*) and two queries from different MPC application areas [79]: the first query (*Password Reuse*) asks for users with the same password across different websites, while the second (*Credit Score*) asks for persons whose credit scores across different agencies have significant discrepancies in a particular year. To showcase the applicability of our optimizations in other domains, we also use three TPC-H queries (**Q4**, **Q6**, **Q13**) that include aggregations along with selections or joins (in Q13 we replace LIKE with an equality since the former is not yet supported by *Secrecy*). Finally, to evaluate the performance gains from each optimization in isolation, we use the three example queries (**Q1**, **Q2**, **Q3**) of Sections 4.2-4.4.

Input data. In all experiments, we use randomly generated tables with 64-bit values. Note that the MPC protocols we use assume a fixed-size representation of shares. The data representation size is implementation-specific and could be increased to any 2^k value without modifying the protocols. We also highlight that using randomly generated 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. No matter whether the input values are real of random, parties compute on shares, which are by definition random.

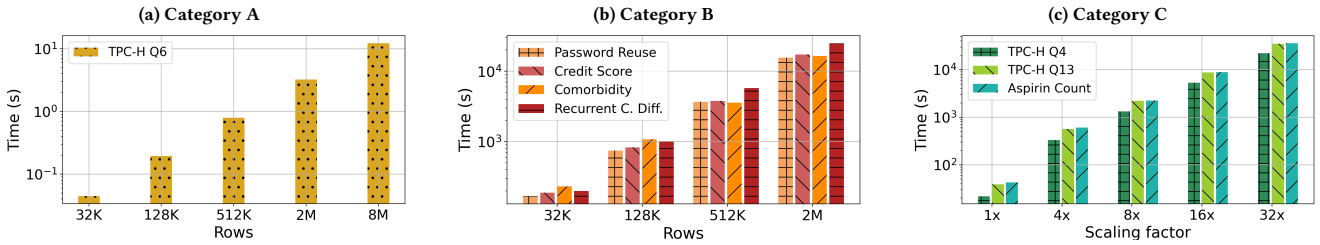


Figure 7: Scaling behavior of optimized real and synthetic queries on *Secrecy*

6.2 Performance on real and synthetic queries

In this section, we evaluate *Secrecy*'s performance on eight queries with and without the optimizations of Section 4. For each query, we implement both the optimized and the non-optimized (baseline) plan using *Secrecy*'s efficient batched operators. Although this favors the baseline, the communication cost of MPC is prohibitive without message batching and queries cannot scale to more than a few hundred input rows in reasonable time.

Comparison with baseline. We execute each query plan with $1K$ rows per input relation and present the results in Figure 6. For *Comorbidity*, we use a cohort of 256 patients. 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.

The optimized plans for *Recurrent C.Diff.*, *Aspirin Count*, and Q13 achieve the highest speedups over non-optimized plans, that is, 1868×, 134×, and 6486× lower execution times respectively. Optimized plans for these queries leverage logical and physical optimizations to push blocking operators before joins (*Aspirin Count*), fuse operators (*Recurrent C.Diff.*), or decompose join with aggregation (Q13). The optimized plans for *Comorbidity*, *Password Reuse*, Q4, and Q6 leverage secret sharing optimizations that result in up to 71× lower execution times compared to non-optimized plans. Finally, the *Credit Score* query leverages dual sharing optimizations, which, in this case, do not provide significant performance improvement.

Scaling behavior. We now run the optimized plans with increasing input sizes and measure 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 7 presents the results.

The only query that falls in *Category A* is Q6. This query includes five selections plus a global aggregation and requires very limited inter-party communication that does not depend on the size of the input relation. As a result, Q6 scales comfortably to large inputs and takes a bit less than $13s$ for $8M$ rows. Queries in *Category B* scale to millions of input rows as well, but with higher execution times compared to Q6. The cost of queries in this category is dominated by the oblivious group-by and distinct operators that rely on oblivious sort. For large inputs, the most expensive of the four queries is *Recurrent C.Diff.*, which completes in $\sim 7h$ for $2M$ input rows.

Finally, queries in *Category C* scale to tens or hundreds of thousands of input rows, depending on the particular operators in the plan. The cost of queries in this category is dominated by the oblivious join and semi-join operators. All three queries have two input relations but with different size ratios: for Q4 and Q13, we use the ratio specified in the TPC-H benchmark whereas for *Aspirin Count* we use inputs of equal size. For each query in Figure 7c, we start with $1K$ rows for the smaller input relation (scaling factor 1×) and increase the size of the two inputs up to 32×, always keeping their ratio fixed. The most expensive query is *Aspririn Count*, as it includes an oblivious θ-join with both equality and inequality predicates. Recall that join needs to perform $O(n \cdot m)$ comparisons, that is, over $1B$ for $64K$ rows ($32K$ per input). Nevertheless, due to *Secrecy*'s ability to push down blocking operators and perform joins in batches, it successfully completes in $\sim 10.5h$. Q4 requires $\sim 6h$ on $164K$ rows, and Q13 is able to complete in $\sim 9.7h$ on $295K$ rows due to the join-aggregation decomposition.

While MPC protocols remain highly expensive for real-time queries, our results demonstrate that offline collaborative analytics on medium-sized datasets entirely under MPC are viable. To the best of our knowledge, *Secrecy* is the first framework capable of evaluating real-world queries on inputs of such scale, while ensuring no information leakage and no reliance on trusted hardware.

6.3 Comparison with other MPC frameworks

In this section, we compare *Secrecy* with two state-of-the-art MPC frameworks: SMCQL [13] and the 2-party semi-honest version of EMP [91]. We choose SMCQL (the ORAM-based version) as the only open-source relational framework with a semi-honest model and no information leakage (cf. Table 1). More recent systems, such as Shrinkwrap [14], SAQE [15], and a new version of SMCQL, although not publicly available, build on top of EMP. Senate [79] also relies on EMP, albeit its malicious version.

Comparison with SMCQL. In the first set of experiments, we aim to reproduce the results presented in the SMCQL paper (Figure 7) [13] on our experimental setup. We run the three medical queries on SMCQL and *Secrecy*, using a sample of 25 tuples 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[4]. As we can see, *Secrecy* is over 2000× faster than SMCQL in all queries, even though SMCQL pushes operators outside the MPC boundary by allowing data owners to execute part of the computation on their plaintext data. In the SMCQL experiment, each computing party is also a data owner and, although it provides 25 tuples per relation to a query, only 8 of those enter the oblivious part of the plan; the rest are filtered out before entering the MPC circuit.

[4]https://github.com/smcql/smcql

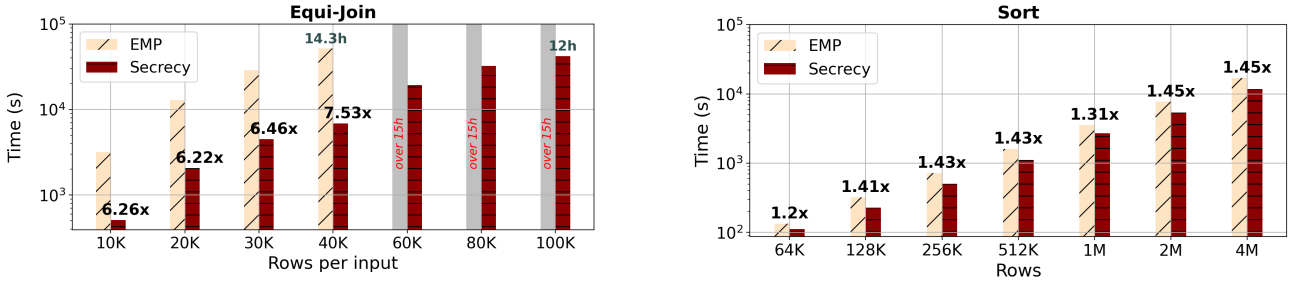
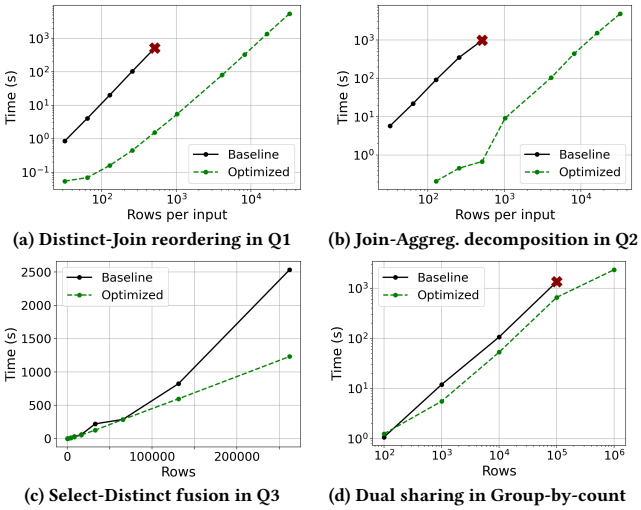


Figure 8: Performance comparison between EMP and *Secrecy* on oblivious equi-join (left) and sort (right). *Secrecy* evaluates the join on $100K$ rows per input in $12h$ whereas EMP requires $14.3h$ for $40K$ rows per input. *Secrecy* is up to $1.45\times$ faster than EMP on oblivious sort.

	Comorbidity	Recurrent C. Diff.	Aspirin Count
SMCQL	197s	804s	796s
Secrecy	0.083s	0.092s	0.171s

Table 4: SMCQL and *Secrecy* execution times for the three medical queries of Section 4.5 on 25 tuples per input relation.



(a) Distinct-Join reordering in Q1

(b) Join-Aggreg. decomposition in Q2

(c) Select-Distinct fusion in Q3

(d) Dual sharing in Group-by-count

Figure 9: Benefits of optimizations on *Secrecy*. Operator reordering (a) and decomposition (b) result in over $100\times$ lower execution times compared to alternative plans, while physical (c) and secret-sharing (d) optimizations improve performance by up to $2\times$. All optimizations enable *Secrecy* to scale to much larger inputs (up to $100\times$) without running out of memory.

Comparison with EMP. EMP is a general-purpose MPC framework and does not provide implementations of relational operators out-of-the-box. For this set of experiments, we implemented an equi-join operator using the sample program available in the SoK project[5] and we also use the oblivious sort primitive provided in the EMP repository[6]. Figure 8 presents the results. For joins, we use inputs of the same cardinality ($n = m$) and increase the size from $10K$ to $100K$ rows per input. We cap the time of these experiments to $15h$. Within the experiment duration, EMP can evaluate joins on up to $40K$ rows per input (in $14.3h$). *Secrecy* is $7.53\times$ faster for the same input size and can process up to $100K$ rows per input in a bit less than $12h$. The performance gap between *Secrecy* and EMP on oblivious sort is less dramatic but still considerable. In this case, both frameworks scale to much larger inputs and *Secrecy* is up to $1.45\times$ faster ($3.27h$ vs $4.74h$ for $4M$ input rows).

6.4 Benefits of optimizations

We now use the example queries of Section 4 (Q1, Q2, Q3) to evaluate the performance impact of *Secrecy*'s optimizations. We run each query with and without the particular optimization and measure total execution time. The results are shown in Figure 9.

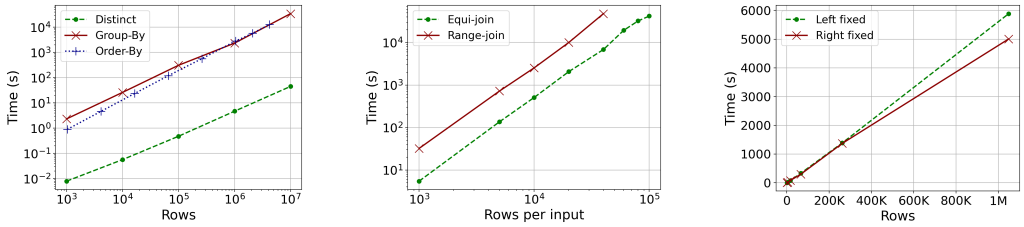
Distinct-Join reordering. Q1 applies `DISTINCT` to the result of an equi-join. The baseline plan executes the oblivious join first, then sorts the materialized cartesian product $R \times S$ and applies `DISTINCT`. In the optimized plan, `DISTINCT` is pushed before the `JOIN` and, thus, *Secrecy* sorts a relation of n rows instead of n^2. Figure 9a shows that the optimized plan is up to two orders of magnitude faster than the baseline, which runs out of memory for even modest input sizes.

Join-Aggregation decomposition. Q2 performs a grouped aggregation on the result of an equi-join. The baseline plan performs the join first, materializes the result, and then applies the grouping and aggregation. Instead, the optimized plan decomposes the aggregation in two phases (cf. Section 4.2.3) and transforms the equi-join into a pipelined semi-join. As shown in Figure 9b, this optimization provides up to three orders of magnitude lower execution time than that of the baseline plan. Further, the materialized join causes the baseline plan to run out of memory for inputs larger than $1K$ rows.

Operator fusion. Q3 applies `DISTINCT` on the result of a selection. The baseline plan applies the oblivious selection and then sorts its output and applies `DISTINCT` sequentially. As we explain in Section 4.3.2, *Secrecy* fuses the two operators and performs the `DISTINCT` computation in bulk. Figure 9c (plot in linear scale) shows that this optimization provides up to $2\times$ speedup for large 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 (Section 4.4) to that of a baseline operator that uses boolean sharing only and, hence, relies on the ripple-carry adder to compute the `COUNT`. Figure 9d plots the

[5]https://github.com/MPC-SoK/frameworks/blob/master/emp/sh_test/test/xtabs.cpp
[6]https://github.com/emp-toolkit/emp-sh2pc



(a) Unary operators (b) Join operators (c) Semi-join operator

Figure 10: Performance of oblivious relational operators in *Secrecy*

results. The baseline operator is 2× slower than the optimized one, as it requires 64 additional rounds of communication per input row.

6.5 Performance of relational operators

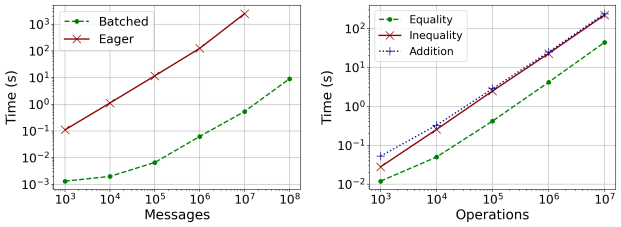
The next set of experiments evaluates the performance of oblivious relational operators in *Secrecy*. We perform `DISTINCT`, `GROUP-BY`, `ORDER-BY`, `IN`, and `JOIN` (equality and range) on relations of increasing size and measure the total execution time per operator. We empirically verify the cost analysis of Section 3 and show that our batched implementations are efficient and scale to millions of input rows with a single thread. Figure 10 shows the results.

Unary operators. In Figure 10a, we plot the execution time of unary operators vs the input size. Recall from Section 3.1 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 10a 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 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 $45s$. `GROUP BY` is slower than `DISTINCT`, as it requires significantly more rounds of communication, linear 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 communication 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 10b. The figure includes equi-join results for up to $100K$ rows per input and range-join results for up to $40K$ rows per input, as we capped the duration of this experiment to $15h$. *Secrecy* executes joins in batches without materializing their entire output at once. As a result, it can perform $10B$ equality comparisons and $1.6B$ inequality comparisons under MPC within the experiment duration limit.

We also run experiments with semi-joins (`IN`) and present the results in Figure 10c. In this case, we vary the left and right input sizes independently, as they affect the cost of the semi-join differently.



(a) Eager vs. batched communication (b) Comparison and addition

Figure 11: Performance of oblivious primitives in *Secrecy*

Each line corresponds to an experiment where we keep one of the inputs fixed to $1K$ rows and increase the size of the other input from $1K$ to $1M$ rows (in powers of two). The two lines overlap when inputs are small (up to $256K$ rows) but they diverge significantly for larger inputs. The reason behind this performance difference is because the number of communication rounds in the semi-join depends only on the size of the right input (cf. Table 2). Although a semi-join between $1M$ (left) and $1K$ (right) rows incurs the same asymptotic number of operations with a semi-join between $1K$ (left) and $1M$ (right) rows, the latter has a higher synchronization cost, which in practice causes a latency increase of $\sim 800s$.

6.6 Micro-benchmarks

To better understand the results of the previous sections, we now use a set of micro-benchmarks and evaluate the performance of *Secrecy*'s MPC primitives.

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 '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.

We run this experiment with increasing data sizes and measure the total time from initiating the exchange until all parties complete the exchange. Figure 11a shows the results. We see that batching provides two to three orders of magnitude lower latency than eager messaging. Using batching in our experimental setup, parties can exchange $100M$ 64-bit data shares in $10s$. 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 11b 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 memory- and computation-intensive as inequality, but requires a higher number of rounds (64).

7 RELATED WORK

Enclave-based approaches. In this line of work, parties apply the oblivious operators on the actual data (rather than secret shares) within a physically-protected environment, such as a trusted server, a hardware enclave or a cryptographic coprocessor. This is a fundamentally different approach to achieve security, where parties send their (encrypted) data to other parties and the oblivious computation happens inside the trusted environment without paying the communication cost of MPC (cf. Table 2). In *Secrecy*, computing parties execute an identical computation on commodity hardware and must communicate multiple times to apply each operator, thus, the main objective is to optimize communication. By contrast, the main objectives within enclave-based approaches are to operate with a small amount of RAM, properly pad intermediate query results, and hide access patterns while reading/writing (encrypted) data from/to untrusted storage. The theoretical works by Agrawal et al. [5] and Arasu et al. [10] focus on secure database queries in this setting. ObliDB [36], Opaque [98], and StealthDB [87] are three recent systems that rely on secure hardware (e.g. Intel's SGX) to support a wide range of database operators, including joins and group-by with aggregation. OCQ [31] builds on Opaque and introduces additional optimizations that reduce intermediate result padding by leveraging foreign-key constraints between private and public relations. Enclave-based systems typically achieve better performance than MPC-based systems but they require different trust assumptions (as an alternative to cryptography) and are susceptible to various side-channel attacks, including branching [65], cache-timing [21, 47, 90], and other attacks [23, 24, 64, 93].

ORAM-based approaches. Oblivious RAM [45, 46] allows for compiling arbitrary programs into oblivious ones by carefully distorting access patterns to eliminate leaks. ORAM-based systems like SMCQL [13] and Obladi [28] hide access patterns but the flexibility of ORAM comes at high cost to throughput and latency. Two-server distributed ORAM systems like Floram [33] and SisoSPIR [56] are faster but require the same non-collusion assumption as in this work. *Secrecy* does not rely on ORAM; instead, we implement specific database operators with a data-independent control flow.

Hybrid query processing. In addition to the frameworks in Table 1, two other works that employ hybrid query execution and let data owners execute as many operators as possible on their plaintext data are those by Aggarwal et al. [4] and Chow et al. [27]. The latter also leverages a semi-trusted party that learns metadata and must not collude with any other party.

Oblivious operators. Related works in the cryptographic and database communities focus on standalone oblivious operators, e.g. building group-by from oblivious sorting [57], building equi-joins [6, 62, 69, 77], or calculating common aggregation operators like `MIN`, `MAX`, `SUM`, and `AVG` [35]. Our work is driven by real-world applications that typically require oblivious evaluation of queries with multiple operators. Two recent works in this direction are [22, 55], however, they focus on specific queries and do not employ any of the optimizations we introduce in this paper.

Outsourced databases. Secure database outsourcing is an active area of research and there are many approaches proposed in the literature. Existing practical solutions [42] use "leaky" cryptographic primitives that reveal information to the database server. Systems based on property-based encryption like CryptDB [80] offer full SQL support and legacy compliance, but each query reveals information that can be used in reconstruction attacks [48, 60, 71]. Systems based on structural encryption [58] like Arx [78], BlindSeer [73], and OSPIR-OXT [25] provide semantic security for data at rest and better protection, but do not eliminate access pattern leaks. SDB [51, 92] uses secret-sharing in the typical client-server model but its protocol leaks information to the database server. KafeDB [97] uses a new encryption scheme that leaks less information compared to prior works. Finally, Cipherbase [9] is a database system that relies on a secure coprocessor (trusted machine).

FHE-based approaches. Fully Homomorphic Encryption (FHE) protocols [43] allow arbitrary computations directly on encrypted data with strong security guarantees. Although many implementations exist [12, 34, 44, 54, 67, 82], this approach is still too computationally expensive for the applications we consider in this work.

Differential privacy. Systems like DJoin [70], DStress [72], and the work of He et al. [50] use the concept of differential privacy to ensure that the output of a query reveals little about any one input record. This property is independent of (yet symbiotic with) MPC's security guarantee that the act of computing the query reveals no more than what may be inferred from its output, and *Secrecy* could be augmented to provide differentially private outputs if desired.

Shrinkwrap [14] and SAQE [15] achieve better efficiency by relaxing security for the computing parties only up to differentially private leakage. This is effectively the same guarantee as above when the computing and result parties are identical, but is weaker when they are different. For this reason, *Secrecy* does not leak anything to computing parties.

MPC frameworks. The recent advances in MPC have given rise to many practical general-purpose MPC frameworks like ABY [32], ABY3 [68], Jiff[20], Obliv-C [96] ObliVM[66], SCALE-MAMBA [63], and ShareMind [18]; we refer readers to Hastings et al. [49] for an overview of these frameworks. Some of these frameworks support standalone database operators (e.g. [11, 18, 68]) but do not address query costs under MPC. Splinter [89] uses function secret sharing to protect private queries on public data. This system supports a subclass of SQL queries that do not include private joins.

8 WHAT'S NEXT?

We see several exciting research directions for the database and systems communities:

MPC query optimizers. Several of our examples showcase that optimal plans in a cleartext evaluation are not necessarily optimal under MPC (and vice versa). Building robust MPC query optimizers that take into account alternative oblivious operators and public information about the data schema is a promising research avenue. The optimizations in Section 4 are by no means exhaustive and there are many opportunities for continued research in this space. For example, Krastnikov et al. [62] and Mohassel et al. [69] recently introduced oblivious algorithms for joins on unique keys with linear (rather than quadratic) worst-case runtime. These algorithms could be extended to avoid materializing intermediate state and applied to other settings like foreign-key joins.

Parallelism and oblivious hashing. Task and data parallelism offer the potential for improved performance and scalability. Extending oblivious operators to work in a task-parallel fashion is straight-forward (e.g. for bitonic sort) but data-parallel execution requires additional care. In a plaintext data-parallel computation, data are often partitioned using hashing: the data owners agree on a hash function f and hash the input records into buckets, so that subsequent join and group-by operations only need to compare records within the same bucket. In MPC, data parallelism can be achieved via oblivious hashing, with care taken to ensure that the bucket sizes do not reveal the data distribution or access patterns. Indeed, many private set intersection algorithms leverage this technique in a setting where the input and computing parties are identical [76]. To achieve better load balancing of keys across buckets and keep the bucket size low, one can use Cuckoo hashing, as in [75, 77]. It is an interesting direction to design oblivious hashing techniques in the outsourced setting, where data owners generate and distribute secret shares along with their corresponding bucket IDs to reduce the cost of oblivious join and group-by operators.

Efficient MPC primitives and HW acceleration. There exist opportunities to improve upon the efficiency of the underlying MPC building blocks used in our operators. First, while we strived to minimize *Secrecy*'s codebase and thus to repurpose oblivious bitonic sort for as many operators as possible, one can achieve even better performance by adding support for more primitives, e.g. a fast oblivious shuffle with linear (rather than quasi-linear) work and constant rounds. Second, while *Secrecy* takes a software-only approach, one could implement special MPC primitives on modern hardware [38–41, 53, 85, 86] to further improve computation and communication latency.

Malicious security. While the current work focuses on semi-honest security, it provides a strong foundation for achieving malicious security in the future. *Secrecy* protects data using the replicated secret sharing scheme of Araki et al. [8], which can be extended to provide malicious security with low computational cost [7]. By optimizing MPC rather than sidestepping it, our approach has an advantage over prior work [79]: we do not need to take additional non-trivial measures to protect the integrity of local pre-processing steps.

ACKNOWLEDGMENTS

The authors are grateful to Kinan Dak Albab, Azer Bestavros, and Ben Getchell for their valuable feedback, and to the Mass Open Cloud for providing access to their cloud for experiments. The fourth author'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] CrypTen. https://github.com/facebookresearch/CrypTen. Last access: January 2021.
[2] Massachusetts Open Cloud. https://massopen.cloud/. Last access: January 2021.
[3] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh Patel, Andrew Pavlo, Raluca Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. 2020. The Seattle Report on Database Research. *SIGMOD Rec.* 48, 4 (Feb. 2020), 44–53. https://doi.org/10.1145/3385658.3385668
[4] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. 2005. Two Can Keep a Secret: A Distributed Architecture for Secure Database Services, In The Second Biennial Conference on Innovative Data Systems Research (CIDR 2005). *CIDR 2005*. http://ilpubs.stanford.edu:8090/659/
[5] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, and Yaping Li. 2006. Sovereign Joins. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, USA, 26. https://doi.org/10.1109/ICDE.2006.144
[6] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. 2003. Information Sharing across Private Databases. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 86âĂŞ97. https://doi.org/10.1145/872757.872771
[7] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. 2017. 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)*. 843–862. https://doi.org/10.1109/SP.2017.15
[8] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (Vienna, Austria). 805–817. https://doi.org/10.1145/2976749.2978331
[9] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security With Cipherbase. In *6th Biennial Conference on Innovative Data Systems Research (CIDR'13)* (6th biennial conference on innovative data systems research (cidr'13) ed.). https://www.microsoft.com/en-us/research/publication/orthogonal-security-with-cipherbase/
[10] Arvind Arasu and Raghav Kaushik. 2014. Oblivious Query Processing. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 26–37. https://doi.org/10.5441/002/icdt.2014.07
[11] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. 2018. From Keys to Databases - Real-World Applications of Secure Multi-Party Computation. *Comput. J.* 61, 12 (2018), 1749–1771.
[12] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex J. Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard W. Ryan. 2019. RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications. In *WAHC@CCS*. ACM, 57–68.
[13] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. 2017. SMCQL: Secure Query Processing for Private Data Networks. *Proc. VLDB Endow.* 10, 6 (2017), 673–684. https://doi.org/10.14778/3055330.3055334
[14] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018), 307–320.
[15] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2691–2705.
[16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. 2016. Students and Taxes: a Privacy-Preserving Study Using Secure

Computation. *Proceedings on Privacy Enhancing Technologies (PoPETS)* 2016, 3 (2016), 117–135. http://www.degruyter.com/view/j/popets.2016.2016.issue-3/popets-2015-0019/popets-2016-0019.xml

[17] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5283)*, Sushil Jajodia and Javier López (Eds.). Springer, 192–206. https://doi.org/10.1007/978-3-540-88313-5_13

[18] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5283)*, Sushil Jajodia and Javier López (Eds.). Springer, 192–206. https://doi.org/10.1007/978-3-540-88313-5_13

[19] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. [n.d.]. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *ACM Conference on Computer and Communications Security*. ACM, 1175–1191.

[20] Boston University. [n.d.]. Javascript Implementation of Federated Functionalities. https://github.com/multiparty/jiff. [Online; accessed September 2020].

[21] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

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

[23] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[24] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1041–1056. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[25] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*. The Internet Society.

[26] Surajit Chaudhuri and Kyuseok Shim. 1994. Including Group-By in Query Optimization. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 354–366. http://www.vldb.org/conf/1994/P354.PDF

[27] Sherman S. M. Chow, Jie-Han Lee, and Lakshminarayanan Subramanian. 2009. Two-Party Computation Model for Privacy-Preserving Queries over Distributed Databases. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society. https://www.ndss-symposium.org/ndss2009/two-party-computation-model-privacy-preserving-queries-over-distributed-databases/

[28] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 727–743.

[29] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. 2016. Confidential Benchmarking Based on Multiparty Computation. In *Financial Cryptography (Lecture Notes in Computer Science, Vol. 9603)*. Springer, 169–187.

[30] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. 2006. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *TCC (Lecture Notes in Computer Science, Vol. 3876)*. Springer, 285–304.

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

[32] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. 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. https://www.ndss-symposium.org/ndss2015/aby---framework-efficient-mixed-protocol-secure-two-party-computation

[33] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *CCS*. ACM, 523–535.

[34] Duality Technologies. [n.d.]. PALISADE. https://gitlab.com/palisade/palisade-release. [Online; accessed September 2020].

[35] F. Emekci, D. Agrawal, A. E. Abbadi, and A. Gulbeden. 2006. Privacy Preserving Query Processing Using Third Parties. In *22nd International Conference on Data Engineering (ICDE'06)*. 27–27.

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

[37] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Found. Trends Priv. Secur.* 2, 2-3 (2018), 70–246.

[38] Xin Fang, Stratis Ioannidis, and Miriam Leeser. 2017. Secure Function Evaluation Using an FPGA Overlay Architecture. In *FPGA*. ACM, 257–266.

[39] Xin Fang, Stratis Ioannidis, and Miriam Leeser. 2019. SIFO: Secure Computational Infrastructure Using FPGA Overlays. *Int. J. Reconfigurable Comput.* 2019 (2019), 1439763:1–1439763:18.

[40] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. 2014. Faster Maliciously Secure Two-Party Computation Using the GPU. In *SCN (Lecture Notes in Computer Science, Vol. 8642)*. Springer, 358–379.

[41] Tore Kasper Frederiksen and Jesper Buus Nielsen. 2013. Fast and Maliciously Secure Two-Party Computation Using the GPU. In *ACNS (Lecture Notes in Computer Science, Vol. 7954)*. Springer, 339–356.

[42] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. 2017. SoK: Cryptographically Protected Database Search. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 172–191. https://doi.org/10.1109/SP.2017.10

[43] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing* (Bethesda, MD, USA) *(STOC '09)*. Association for Computing Machinery, New York, NY, USA, 169âĂŞ178. https://doi.org/10.1145/1536414.1536440

[44] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *Advances in Cryptology – EUROCRYPT 2011*, Kenneth G. Paterson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 129–148.

[45] O. Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87)*. Association for Computing Machinery, New York, NY, USA, 182âĂŞ194. https://doi.org/10.1145/28395.28416

[46] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431âĂŞ473. https://doi.org/10.1145/233551.233553

[47] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security* (Belgrade, Serbia) *(EuroSec'17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3065913.3065915

[48] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *CCS*. ACM, 315–331.

[49] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *IEEE Symposium on Security and Privacy*. IEEE, 1220–1237.

[50] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. 2017. Composing Differential Privacy and Secure Computation: A Case Study on Scaling Private Record Linkage. In *ACM Conference on Computer and Communications Security*. ACM, 1389–1406.

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

[52] Adrian F. Hernandez, Rachael L. Fleurence, and Russell L. Rothman. 2015. The ADAPTABLE Trial and PCORnet: Shining Light on a New Research Paradigm. *Ann Intern Med.* 163, 8 (2015), 635–636. https://doi.org/10.7326/M15-1460

[53] Siam U. Hussain, Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar. 2018. MAXelerator: FPGA accelerator for privacy preserving multiply-accumulate (MAC) on cloud servers. In *DAC*. ACM, 33:1–33:6.

[54] howpublished = "https://github.com/homenc/HElib" note = "[Online; accessed September 2020]" IBM Research, title = HElib. [n.d.].

[55] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. 2019. On Deploying Secure Computing Commercially: Private Intersection-Sum Protocols and their Business Applications. *IACR Cryptology ePrint Archive* 2019 (2019), 723.

[56] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2016. Private Large-Scale Databases with Distributed Searchable Symmetric Encryption. In *CT-RSA (Lecture Notes in Computer Science, Vol. 9610)*. Springer, 90–107.

[57] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. 2011. Secure Multi-Party Sorting and Applications. *IACR Cryptol. ePrint Arch.* 2011 (2011), 122. http://eprint.iacr.org/2011/122

[58] Seny Kamara and Tarik Moataz. 2018. SQL on Structurally-Encrypted Databases. In *Advances in Cryptology – ASIACRYPT 2018*, Thomas Peyrin and Steven Galbraith (Eds.). Springer International Publishing, Cham, 149–180.

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

[60] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1329âĂŞ1340. https://doi.org/10.1145/2976749.2978386

[61] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA.

[62] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proc. VLDB Endow.* 13, 11 (2020), 2132–2145. http://www.vldb.org/pvldb/vol13/p2132-krastnikov.pdf

[63] KU Leuven. [n.d.]. SCALE-MAMBA Software. https://homes.esat.kuleuven.be/~nsmart/SCALE/. [Online; accessed September 2020].

[64] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 487–504. https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol

[65] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 557–574. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[66] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)* (San Jose, California, USA). 359–376. https://doi.org/10.1109/SP.2015.29

[67] Microsoft Research. [n.d.]. SEAL. https://github.com/Microsoft/SEAL. [Online; accessed September 2020].

[68] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 35âĂŞ52. https://doi.org/10.1145/3243734.3243760

[69] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins for Secret Shared Data.

[70] Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially Private Join Queries over Distributed Databases. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (Hollywood, California, USA). 149–162. http://dl.acm.org/citation.cfm?id=2387880.2387895

[71] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *ACM Conference on Computer and Communications Security*. ACM, 644–655.

[72] Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. 2017. DStress: Efficient Differentially Private Computations on Distributed Data. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)* (Belgrade, Serbia). 560–574. https://doi.org/10.1145/3064176.3064218

[73] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. 2014. Blind Seer: A Scalable Private DBMS. In *2014 IEEE Symposium on Security and Privacy*. 359–374.

[74] Patient-Centered Outcomes Research Institute (PCORI). 2015. Characterizing the Effects of Recurrent Clostridium Difficile Infection on Patients. IRB Protocol, ORA: 14122.

[75] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2020. PSI from PaXoS: Fast, Malicious Private Set Intersection. In *EUROCRYPT (2) (Lecture Notes in Computer Science, Vol. 12106)*. Springer, 739–767.

[76] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. 2015. Phasing: Private Set Intersection Using Permutation-based Hashing. In *USENIX Security Symposium*. USENIX Association, 515–530.

[77] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. 2018. Efficient Circuit-Based PSI via Cuckoo Hashing. In *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 (Lecture Notes in Computer Science, Vol. 10822)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, 125–157. https://doi.org/10.1007/978-3-319-78372-7_5

[78] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: An Encrypted Database Using Semantically Secure Encryption. *Proc. VLDB Endow.* 12, 11 (July 2019), 1664âĂŞ1678. https://doi.org/10.14778/3342263.3342641

[79] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Vancouver, B.C. https://www.usenix.org/conference/usenixsecurity21/presentation/poddar

[80] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal). 85–100. https://doi.org/10.1145/2043556.2043566

[81] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. Washington, DC, USA, 655–670. https://doi.org/10.1109/SP.2014.48

[82] Kurt Rohloff and David Bruce Cousins. 2014. A Scalable Implementation of Fully Homomorphic Encryption Built on NTRU. In *Financial Cryptography and Data Security*, Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 221–234.

[83] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. https://doi.org/10.1145/359168.359176

[84] Jung Hoon Song and You Sun Kim. 2019. Recurrent Clostridium difficile Infection: Risk Factors, Treatment, and Prevention. *Gut and liver* 13, 1 (2019), 16–24. https://doi.org/10.5009/gnl18071

[85] Ebrahim M. Songhori, M. Sadegh Riazi, Siam U. Hussain, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2019. ARM2GC: Succinct Garbled Processor for Secure Computation. In *DAC*. ACM, 112.

[86] Ebrahim M. Songhori, Shaza Zeitouni, Ghada Dessouky, Thomas Schneider, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2016. GarbledCPU: a MIPS processor for secure computation in hardware. In *DAC*. ACM, 73:1–73:6.

[87] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 370–388. https://doi.org/10.2478/popets-2019-0052

[88] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 3:1–3:18. https://doi.org/10.1145/3302424.3303982

[89] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. 2017. Splinter: Practical Private Queries on Public Data. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 299âĂŞ313.

[90] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. 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* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2421âĂŞ2434. https://doi.org/10.1145/3133956.3134038

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

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

[93] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 640âĂŞ656. https://doi.org/10.1109/SP.2015.45

[94] Weipeng P. Yan and Per-Åke Larson. 1994. Performing Group-By before Join. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*. IEEE Computer Society, 89–100. https://doi.org/10.1109/ICDE.1994.283001

[95] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*. IEEE Computer Society, USA, 162âĂŞ167. https://doi.org/10.1109/SFCS.1986.25

[96] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. arXiv:2015/1153 http://eprint.iacr.org/2015/1153.

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

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