Making Privacy-preserving Federated Graph Analytics Practical (for Certain Queries)

Kunlong Liu
University of California, Santa Barbara
Santa Barbara, CA, USA
kunlongliu@ucsb.edu

Trinabh Gupta
University of California, Santa Barbara
Santa Barbara, CA, USA
trinabh@ucsb.edu

ABSTRACT

Privacy-preserving federated graph analytics is an emerging area of research. The goal is to run graph analytics queries over a set of devices that are organized as a graph while keeping the raw data on the devices rather than centralizing it. Further, no entity may learn any new information except for the final query result. For instance, a device may not learn a neighbor's data. The state-of-the-art prior work for this problem provides privacy guarantees for a broad set of queries in a strong threat model where the devices can be malicious. However, it imposes an impractical overhead. For example, for a certain query, each device locally requires over 8.79 hours of CPU time and 5.73 GiBs of network transfers. This paper presents Colo, a new, low-cost system for privacy-preserving federated graph analytics that requires minutes of CPU time and a few MiBs in network transfers, for a particular subset of queries. At the heart of Colo is a new secure computation protocol that enables a device to securely and efficiently evaluate a graph query in its local neighborhood while hiding device data, edge data, and topology data. An implementation and evaluation of Colo shows that for running a variety of COVID-19 queries over a population of 1M devices, it requires less than 8.4 minutes of a device's CPU time and 4.93 MiBs in network transfers—improvements of up to three orders of magnitude.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**.

KEYWORDS

federated analytics; graph queries; secure multi-party computation; zero-knowledge proof

ACM Reference Format:
Kunlong Liu and Trinabh Gupta. 2024. Making Privacy-preserving Federated Graph Analytics Practical (for Certain Queries). In *Proceedings of the 29th ACM Symposium on Access Control Models and Technologies (SACMAT 2024), May 15–17, 2024, San Antonio, TX, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3649158.3657047

1 INTRODUCTION

As a motivating example, consider the following scenario between a mobile app maker of a contact tracing application for an infectious disease like COVID-19 [33], and an influential data analyst such as the Centers for Disease Control and Prevention (CDC) [5]. The app maker installs the app on a large number of mobile devices, where it collects information on whether a device owner is currently infected, and when, where, and for how long the device comes in contact with other devices. Abstractly, one can view the devices as a graph where they are the nodes and their interactions are the edges. Meanwhile, the analyst wants to use the data to study disease patterns. For instance, it wants to understand the prevalence of superspreaders by evaluating the average number of infected devices in an infected device's neighborhood [16, 28]. *Can we enable the analyst to run such queries and learn their result? Further, can we do it in a way that doesn't require moving device and edge data outside the devices to a centralized location? And further still, can we ensure that only the query result is revealed and no new individual device information is learned by any other party?*

This is the problem of *privacy-preserving federated graph analytics*. The federated aspect of this problem emphasizes keeping raw data at the devices, in contrast to centralizing the data, which is highly susceptible to data breaches, especially in bulk [30, 34]. Meanwhile, the privacy guarantee of the problem emphasizes that an entity should get only the information that it absolutely needs. Thus, an analyst may learn the query result that is an aggregate across devices. And any device may not learn any more information than it knows locally through its own data and edges.

As we discuss in related work (§7), privacy-preserving federated graph analytics is an emerging area of research and displays a trade-off between generality, privacy, and efficiency. For instance, Gunther et al. [13] built a system RIPPLE to answer epidemiological questions. However, their system answers aggregation queries only where a device can securely sum its state (e.g., an integer) with its neighbors' state, but not other secure operations. Thus, it cannot answer our example query on superspreaders, which also requires computing multiplications. In contrast, Roth et al. [31] have built a general-purpose system, Mycelium, that assumes a strong threat model where both the devices and a centralized aggregator can be malicious. However, Mycelium is expensive. For the superspreader query over 1M devices, each device incurs 8.79 hours of local CPU time and 5.73 GiB of network transfers.

This paper introduces Colo, an efficient system for privacy-preserving federated graph analytics. Colo allows an analyst to run simple queries (like the superspreader query) that have predicates with a limited set of inputs and outputs, and that evaluate these predicates between a device and its neighbors. Colo guarantees privacy (only the analyst learns the query result and no entity gets any other intermediate data) while assuming malicious devices and a set of M aggregation servers of which a fraction f can be

SACMAT 2024, May 15–17, 2024, San Antonio, TX, USA

ACM ISBN 979-8-4007-0491-8/24/05
https://doi.org/10.1145/3649158.3657047

malicious. Finally, Colo is scalable and efficient: it supports a large number of devices in the order of a few million, while requiring them to contribute a small amount of CPU and network.

At a high level, Colo follows a workflow of local evaluation followed by a global aggregation across devices (§3). In the local evaluation, each device evaluates the query between itself and its neighbors. The global aggregation then aggregates these per-device outputs. In this workflow, Colo must address two challenges. First, it must hide node, edge, and topology data during the local evaluation without imposing a large overhead on the devices. Second, it must aggregate per-node outputs across the population of devices without revealing the intermediate results.

The first challenge of hiding node, edge, and topology data is tricky, especially with malicious devices. For instance, say two neighboring devices v_A and v_B want to compute the product $v_A.inf \cdot v_B.inf$ (as needed for the superspreader query), where *inf* indicates their infection status. Then, a malicious device, say v_B, may set its infection status to $v_B.inf = 10^8$. As a result, the query result secretly encodes v_A's infection status (result is large if $v_A.inf = 1$). One may use a general-purpose tool from cryptography to address this issue, but that would be expensive. Furthermore, even if there were an efficient protocol for this computation, say that requires a single interaction between v_A and v_B, then this protocol must also hide that v_A and v_B are communicating, to protect their topology data, i.e., the fact they are neighbors.

Colo addresses this first challenge through a new, tailored secure computation protocol (§4.3.1). Colo observes that the query predicates that it targets operate over a limited set of inputs and produce a limited set of outputs. For instance, the legitimate inputs and outputs for $v_A.inf \cdot v_B.inf$ are all either zero or one. Thus, instead of using a general purpose secure computation protocol such as Yao's Garbled Circuits [36] that operates over arbitrary inputs and outputs, Colo uses a protocol that operates over a limited set of inputs and outputs. Specifically, one party, say v_B, computes all possible legitimate query outputs in plaintext, and then allows the other party v_A to pick one of these outputs privately using oblivious transfers (OT) [6, 29]. Colo fortifies this protocol against malicious behavior of v_B by incorporating random masks, efficient commitments [3, 10], and range proofs [9, 11] (§4.3.1).

The protocol described above doesn't yet address the requirement of hiding the topology of devices. To hide this data efficiently, that is, the knowledge of who is a neighbor with whom, Colo prohibits devices from directly interacting with each other. Rather, Colo arranges for them to communicate via a set of servers, specifically, a set of 40 to 100 servers, where up to $f\%$ (set to 20% in our experiments) are malicious (§3). This arrangement enables the servers to run a particular metadata hiding communication system, Karaoke [17], that is provably secure and low cost for the devices (although with significant overhead for the servers) (§4.3.2).

Colo addresses the second challenge of global aggregation across devices through straightforward secret sharing techniques while piggybacking on the set of servers (§3, §4.4). Specifically, devices add zero sum masks to their local outputs, and send shares of these local results to the servers. As long as one of the servers is honest, the analyst learns only the query output.

We implemented (§5) and evaluated (§6) a prototype of Colo. Our evaluation shows that for 1M devices connected to at most 50 neighbors each, and for a set of example queries (Figure 1) which includes the superspreader query, a device in Colo incurs less than 8.4 minutes of (single core) CPU time and 4.93 MiB of network transfers. In contrast, the Mycelium system of Roth et al. [31] requires a device-side cost of 8.79 hours (single core) CPU time and 5.73 GiB network transfers. In addition, Colo's server-side cost, depending on the query, ranges from $158 to $1,504 total for Colo's 40 servers (the lower number is for the superspreader query). In contrast, Mycelium's server-side cost is over $57,490 per query.

Colo's limitations are substantial. In particular, it does not handle general purpose queries, rather only those that evaluate predicates over a bounded set of inputs and outputs. However, Colo scales to a significant number of devices and is efficient for them, in a strong threat model. But more importantly, unlike prior work, Colo shows that privacy-preserving federated graph analytics can be practical, and that the CDC *could* run certain queries over the devices' data while guaranteeing privacy in a strong sense, without draining the devices' resources, and without aggressively depleting its own budget (e.g., running the superspreader query in a large city every two weeks would cost around four thousand dollars annually).

2 PROBLEM STATEMENT

2.1 Scenario

We consider a scenario consisting of a data analyst $\mathcal{A}$ and a large number of mobile devices v_i for $i \in [0, N)$. For instance, $N = 10^6$.

The devices form a graph. As an example, they may run a contact tracing application for COVID-19 [33] that collects information on the infection status of the device owners and identities of the devices they come in contact with, that is, their neighbors. More precisely, a device may have (i) *node data*, for example, a status variable *inf* indicating whether the device's owner is currently infected, *tInf* indicating the time the owner got infected (or null if the owner is not infected), and demographic information such as age and ethnicity; (ii) *edge data*, for example, the number of times the device came in contact with a neighbor (*contacts*), the cumulative duration (*duration*) of these interactions, and (*location*, *time*, *duration*) of each interaction; and, (iii) *topology data*, for example, the list of the device's neighbors.

The analyst $\mathcal{A}$, say a large hospital or CDC in the context of the contact tracing application, wants to analyze the device data by running graph queries. Figure 1 shows a few example queries from the literature [2, 16, 24, 25, 28] (these queries are a subset of the ones considered in the Mycelium system of Roth et al. [31]). For instance, $\mathcal{A}$ may want to learn the number of active infections infected devices have in their neighborhood (Q1). These queries perform a local computation at every device and its neighborhood, and then aggregate the per-device results.

2.2 Threat model

We assume that the devices are malicious, i.e., an adversary can compromise a subset of devices. A compromised device may try to learn information about an honest device beyond what it knows from its own data. For instance, if a compromised device is a neighbor of an honest device, then the compromised device already has edge data for their edge (e.g., the location and time of their last meeting); however, the adversary may further want to learn if the

Query	Description
Q1	The total number of infections in an infected participant's neighborhood *SELECT COUNT(*) FROM neigh(1) WHERE self.inf & neighbor.inf*
Q2	The amount of time neighbor has spent near infected device if neighbor is infected within 5-15 days of contact with the device *SELECT SUM(edge.duration) FROM neigh(1) WHERE self.inf & neighbor.inf & (neighbor.tInf ∈ [edge.lastContact+5 days, edge.lastContact+15 days])*
Q3	The frequency of contact between device and neighbor, if device infected neighbor *SELECT SUM(edge.contacts)/COUNT(*) FROM neigh(1) WHERE self.inf & neighbor.inf & (neighbor.tInf > self.tInf+2 days)*
Q4	Secondary attack rate of infected devices if they traveled on the subway *SELECT SUM(neighbor.inf)/COUNT(*) FROM neigh(1) WHERE self.inf & onSubway(edge.lastContact.location)*
Q5	The number of secondary infections caused by infected devices in different age groups *SELECT COUNT(*) FROM neigh(1) WHERE self.inf & neighbor.inf & (neighbor.tInf > self.tInf+2 days) GROUP BY self.age*
Q6	The number of secondary infections based on type of exposure (such as family, social, work) *SELECT COUNT(*) FROM neigh(1) WHERE self.inf & neighbor.inf & (neighbor.tInf > self.tInf+2 days) GROUP BY edge.setting*
Q7	Secondary attack rates in household vs non-household contacts *SELECT SUM(neighbor.inf)/COUNT(*) FROM neigh(1) WHERE self.inf GROUP BY isHousehold(edge.lastContact.location)*
Q8	Secondary attack rates within case-contact pairs in the same age group *SELECT SUM(neighbor.inf)/COUNT(*) FROM neigh(1) WHERE self.inf & neighbor.age∈[0,100] & self.age∈[neighbor.age-10,neighbor.age+10]*

Figure 1: **Example graph queries from Mycelium [31] and the literature on health analytics [2, 8, 12, 15, 16, 24, 25, 28]. We assume that the domain of the inputs to these queries is bounded, for example,** $inf \in [0, 1]$ **and** $tinf \in [1, 120]$**, referring to the days in the latest few months.**

honest device is infected (neighbor's node data), whether it recently met someone on the subway (neighbor's edge data), and whom it recently came in contact with (neighbor's topology data).

The analyst may also be malicious and want to learn about individual device node, edge, or topology data—information that is more granular than the result of the queries.

Finally, we assume that the adversary may also observe and manipulate network traffic, for instance, in the backbone network, and try to infer relationships between devices.

2.3 Goals and non-goals

Target queries. Ideally, we would support arbitrary graph queries. However, as noted earlier (§1), generality of queries is in tension with privacy and efficiency. Thus, in this paper we focus on simpler queries such as those in Figure 1 where the aggregations across devices are SUM, COUNT and AVG operations, and where devices compute simple predicates on a small set of possible inputs in their one hop neighborhood (for example, the infection status *inf* is either zero or one, and the time of infection *tinf* is in $[1, 120]$ referring to the days in the latest few months). Although these queries are a sub-class of a broad set of queries, they are important according to the health literature and form a precursor to more sophisticated queries in an analyst's workflow.

Privacy (P1). Private data should always be hidden from the adversary. From the point of view of a device, it may not learn any information beyond its own node, edge, and topology data. In particular, it may not learn any information about a neighbor beyond what is contained in the direct edge to the neighbor. Similarly, the analyst must only learn the query result.

Privacy (P2). Individual devices will cause bounded changes to query result in the presence of malicious neighbors and servers. This is not a formal privacy guarantee but can be easily extended to differential privacy by adding noise to the query result. In this work, we consider bounded contribution as our privacy goal. For example, in certain counting queries, each device should contribute at most 1 to the final result.

Scale and efficiency. First, our system must support a large number of devices, for example, one million. We assume that these devices have a bounded degree, for example, up to 50 neighbors. (If the actual device graph has nodes with a larger degree, then they may select a subset uniformly for the query execution.) Second, we want device overhead to be low, for example, minutes of CPU time and at most a few MiBs in network transfers, which is affordable for mobile devices. Meanwhile, if the system employs any servers, e.g., cloud servers, then we want the server-side cost to be affordable. In terms of dollar cost of renting the servers, we want the per-server cost to be no more than a few tens of dollars per query over a million devices.

Correctness (non-goal). In this paper, we focus on privacy and require the query output to be correct only during periods when all parties (devices and any servers) are semi-honest. Ensuring correctness in the periods of malicious behavior is likely to be very expensive, and we leave it as future work. Nevertheless, the system must ensure privacy at all times.

3 OVERVIEW OF COLO

Colo takes a federated approach where each device keeps its raw data local and first communicates with its neighbors to perform local computation (e.g., aggregation at the neighborhood level), before uploading this local result to a server for aggregation. The federated approach has distinct advantages relative to the centralized approach: (i) it allows devices to feel more trust in the system as they keep possession of their data, (ii) it doesn't risk bulk data breaches as devices do not collectively centralize their data, and (iii) it handles data updates naturally as each device computes locally and can use its latest data.

Colo relies on a set of servers, for example, 40-100 servers, S_i for $i \in [0, M)$ (Figure 2). Colo assumes that at most f% of these servers may be compromised by the adversary; our implementation sets $f = 20$%. That is, an adversary may compromise, e.g., up to 8 servers when the number of servers is 40. M and f are both configurable. In

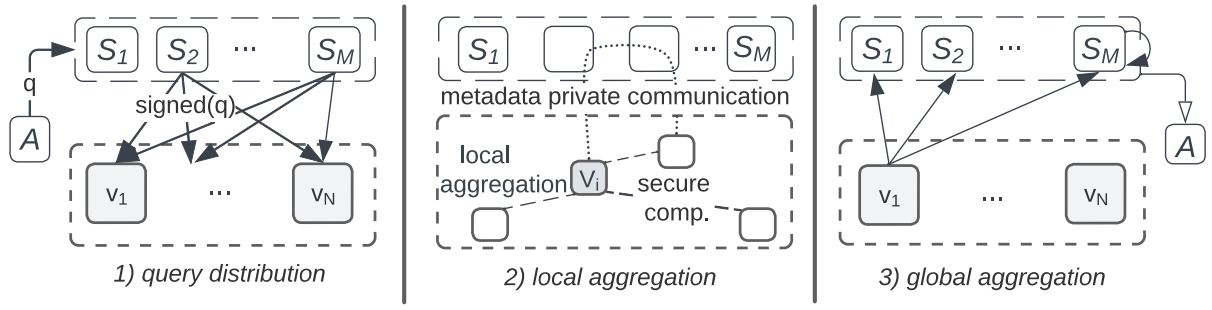


Figure 2: **An overview of Colo's query distribution, local aggregation, and global aggregation phases of query execution. The dotted line in local aggregation depicts metadata-hiding communication, and the dashed line depicts secure computation.**

Colo, devices do not directly interact with each other; rather, they communicate via the servers.

Colo has a one-time *setup* phase, and three phases of *query distribution*, *local aggregation*, and *global aggregation* for query execution (Figure 2).

In the setup phase (not shown in the figure), the servers generate and distribute keys for a cryptographic protocol used in the local aggregation phase. Specifically, they generate the proving and verification keys for a zero-knowledge proof (ZKP) scheme [37].

In the query distribution phase (the leftmost diagram in Figure 2), the analyst $\mathcal{A}$ specifies the query, say q, and sends it to each of the servers. Since $\mathcal{A}$ can be malicious (§2.2) and may write a query that tries to infer a single device's data, the servers validate $\mathcal{A}$'s query. For example, if q contains a WHERE clause of the form *WHERE self.ID = xxx*, then the servers reject it. Colo assumes that the servers have a list of certified queries that are allowed. Once each server validates the query, it signs and broadcasts it to the devices. A device starts the next phase after validating enough signatures, that is, more than a threshold of servers that can be compromised.

In the local aggregation phase (the center diagram in Figure 2), each device evaluates the query in its neighborhood. For instance, for the query Q1 (Figure 1), each node computes the local count of infected neighbors in its neighborhood. Thus, a device uses a secure computation protocol with its neighbor such that at the end of the protocol the two parties receive *secret shares* of the result of the computation. This protocol admits malicious devices; for instance, a malicious neighbor is prevented from supplying an arbitrary input such as setting its $inf = 10^6$.

Secure computation hides node and edge data; however, an adversary that observes network traffic can infer topology by monitoring who is performing secure computation with whom. Thus, the devices in the local aggregation phase execute secure computation over a metadata-hiding communication network, particularly, the Karaoke system [17]. The servers facilitate and run this system.

Finally, in the global aggregation phase (the right diagram in Figure 2), the devices send their results from the local aggregation to the servers, who aggregate them. Specifically, each device secret shares its result with the servers, who locally add the shares they receive from the devices. The servers send the result of their local computation to the analyst who combines these outputs across the servers to obtain the query result.

4 DESIGN DETAILS

This section describes the details of Colo. Figure 3 provides a high-level overview and how the phases connect with each other.

4.1 Setup (key generation)

As we will describe later (§4.3.1), devices in Colo's local aggregation phase need to generate an array T of values and prove that each value is within a range $[L, U]$. The challenge is that the efficient and popular zero-knowledge proof (ZKP) schemes such as that of Groth [11] bind the proving and verification keys to the statement (the circuit) the prover is proving. That is, this circuit is a function of the length $len(T)$ of array T in our case. Unfortunately, the $len(T)$ further depends on the analyst's query q, and may be different for different queries.

To reuse the keys for different queries, Colo generates a key set $\{key_{2^0}, key_{2^1}, \ldots, key_{2^{\log(len(T))-1}}\}$ containing $\log(len(T))$ keys for all powers of two between 1 and anticipated maximum length $len(T)$, for example, 1024. In this way, the size of the key set is manageable, e.g., 102 MiB, and during query execution devices generate no more than $\log(len(T))$ proofs.

4.2 Query distribution

As mentioned in the overview (§3), the analyst $\mathcal{A}$ specifies a query q and gives it to the servers, who validate it. Here, we elaborate on how $\mathcal{A}$ specifies the query.

Each query has two components: (i) a SQL query similar to the examples we have discussed (Figure 1), and (ii) a transformation function, *PreProcess*, that transforms the raw data at a device into a form needed by the SQL query.

As noted earlier (§2.3), Colo does not support arbitrary SQL queries. Rather, it targets simple queries of the form *SELECT AGG-OP(g(self.data, neighbor.data)) FROM neigh(1) WHERE h(self.data, neighbor.data)*, where AGG-OP is the SUM, COUNT, or AVG aggregation operation, g is a predicate on the data of two neighbors (including their node and edge data), and h is a filter that runs over the same data to compute whether a particular edge will participate in aggregation or not. If $\mathcal{A}$ wants to run a query with a GROUP BY operation, $\mathcal{A}$ transforms it into several sub-queries and submits them to the servers separately. For example, $\mathcal{A}$ converts Q5 in Figure 1 into a series of queries such as *SELECT COUNT(*) FROM neigh(1) WHERE 20 < self.age < 30* for the different age groups.

Setup (one time)

- The servers S_i, $i \in [0, M)$, generate a set of keys for a zero-knowledge proof (ZKP) scheme using a MPC protocol [37].
- The servers distribute the keys to the devices. A device downloads the set of keys from one server and hashes of these keys from the others to defend against attacks from malicious servers (a malicious server can distribute malicious keys). The devices stores the keys locally.

Query distribution

(1) The analyst $\mathcal{A}$ submits a query q to all the servers.

(2) The servers verify the query and sign it. Each server broadcasts its signed query to all devices to start query execution.

Local aggregation

(3) The servers run the Karaoke system [17] to enable the devices to communicate anonymously with each other.

(4) Each device initializes its local result of query execution to 0 and runs the local aggregation secure computation protocol (§4.3.1; Figure 4) with its neighbors over the Karaoke system [17] to obtain a share of its local result.

Global aggregation

(5) Each device secret shares its own share of its local result with the servers.

(6) Each server sums all secret shares it receives. Each server sends its sum to the analyst and the analyst adds the sum across the servers to reconstruct the global result.

Figure 3: **A high-level description of Colo's phases.**

The *PreProcess* function specifies how a device should translate its raw data into the attributes accessed by the SQL query. Along with encoding, constraints (bounds) are necessary as otherwise malicious devices can supply arbitrary inputs. For example, to execute Q3 in Figure 1, the analyst needs to specify i) device infection status *inf* as a binary number; ii) the infection timestamp *tInf* as an integer from 1 to 30 to represent March 1 to 30; iii) *edge.contacts*, which is the number of interactions two neighbors have had, as a bounded integer, for example, 80 as in the epidemiology literature [16].

Once the servers receive $\mathcal{A}$'s query, they verify it by matching it to a list of certified queries. The servers then sign and broadcast the query; if a device verifies signatures from more than the fraction of servers that can be compromised, it starts query execution.

4.3 Local aggregation

4.3.1 Hiding node and edge data. Recall from the overview (§3) that the goal of local aggregation is to enable a device to compute the query locally just on its neighborhood. This computation further breaks into evaluating the query for every neighbor edge of a node. That is, a node needs to evaluate a function $F = g \circ h(self.data, neighbor.data)$ with each of its neighbors and compute $\sum_{neighbor} F(self.data, neighbor.data)$. As an example, for query Q1 the function F equals $F(self.data, neighbor.data) = self.inf \cdot neighbor.inf$.

For the moment, assume that we do not need to hide the topology data at the devices (we will relax this assumption in the next subsection). Then, a natural option for computing F is to use a two-party secure computation protocol such as Yao's garbled circuit (Yao's GC) [35]. The neighbor, say, v_B, could act as the garbler, generate a garbled circuit, send it to the other node, say, v_A, who would act as the evaluator to obtain the result. Since the two nodes must not obtain the result of F in plaintext, the two nodes may compute $y = F(r_{v_A}, r_{v_B}, v_A.inf, v_B.inf) = (v_A.inf \cdot v_B.inf) + r_{v_A} + r_{v_B}$, where $r_{v_A}, r_{v_B} \in \mathbb{F}$ are uniformly sampled masks supplied by the two parties to hide the output from each other. (The field $\mathbb{F}$ could be 2^{64}, for example.) At the end of the protocol, v_B may store $-r_{v_B}$ as its output, while v_A may store $-r_{v_A} + y$ as its output.

The challenge is in preventing malicious behavior efficiently. To protect against a malicious neighbor (garbler v_B), Colo would have to use a version of Yao's GC that employs techniques such as cut-and-choose [18, 19, 23] that prevent a garbler from creating arbitrary circuits, for example, an F' that computes, $(v_A.inf \cdot 10^6) + r_{v_A} + r_{v_B}$. These general-purpose primitives are expensive because they are not tailored for the queries and they offer more than we need: malicious integrity is not our goal (§2.3).

Colo observes that the predicates F that appear in Colo's target queries have bounded inputs and outputs. For example, Q1 has two possible inputs of 0 and 1 for $v_A.inf$ and thus two possible outputs of 0 and 1. As another example, if we take Q3 in Figure 1 and assume *tInf* has 30 possibilities (for 30 days), then the number of possible inputs for v_A is sixty. That is, v_A's input pair $(inf, tInf)$ can range from the case $(0, 0)$ to the case $(1, 29)$. Corresponding to each of these inputs, the output *edge.contacts* may be a value in the range $[0, 79]$.

Leveraging this observation, the neighbor in Colo (v_B above) *precomputes* outputs for all possible inputs of v_A into an array T instead of generating them at *runtime* inside Yao's GC. One can view this arrangement as making v_B commit to the outputs without looking at $v_A's$ input. For instance, for Q3, v_B would generate a T of length 60 where each entry is in the range $[0, 79]$. Then, v_B can arrange for v_A to get one of these values obliviously.

Protocol details. Figure 4 shows the details of Colo's protocol. For a moment, assume that the nodes v_A and v_B are honest-but-curious. Then, for every possible input of v_A, the neighbor v_B generates one entry of array T. It also adds a mask, $r \in \mathbb{F}$, to each entry of T. That is, after generating the array T, v_B offsets each entry by the same mask and computes $T'[i] = T[i] + r, i \in [0, len(T))$, where r is private to v_B. The node v_A then obtains one of the entries of the table corresponding to its input using 1-out-of-$len(T)$ oblivious transfer [6, 29]. Finally, v_A uploads $T[j] + r$ for the global aggregation, and v_B uploads $-r$ to cancel the mask.

Local aggregation protocol of Colo

(1) Each device receives a query q containing a function *PreProcess*, a local computation F, and a bound *Bound*. *PreProcess* contains information on how to convert a device's raw data into inputs for F. It also specifies the set of valid values for each input parameter to F. The bound *Bound* is the range of valid outputs of F.

(2) Each device participates in this local aggregation protocol. Denote A as the device and B as a neighbor.

(3) Each neighbor B of A does the following locally:

  (a) (Generate mask) Samples an element r in a field $\mathbb{F}$ uniformly randomly.
  (b) (Enumerate all inputs of A) Generates an array s containing all possible inputs of A based on *PreProcess*.
  (c) (Compute outputs) For every $s[i]$, computes $T[i] = F(s[i], B_{in})$, where B_{in} is B's input.
  (d) (Mask outputs and commit to them) For every $T[i]$, computes $T'[i] = T[i] + r$. It then samples $R[i]$ and generates commitment $CM[i] = Commit(T'[i], R[i])$.
  (e) (Prove bound of outputs) Generates a ZKP that it knows the opening of all commitments and a mask such that each committed value is the addition of the mask and some value bounded by *Bound*.

(4) Each neighbor B sends commitments CM and the ZKP to A which verifies the ZKP.

(5) (Function evaluation) A runs OT with each neighbor B to retrieve $(T'[j], R[j])$, where $s[j]$ is A's input to F. The device A verifies the opening to the commitment, that is, it verifies $Commit(T'[j], R[j])$ equals $CM[j]$ received in the previous step.

(6) If the verifications above pass, A adds $T'[j]$ to its local aggregation result. Its neighbor device B adds $-r$ to its local aggregation result.

Figure 4: **Colo's local aggregation.**

To account for a malicious neighbor v_B who tries to break our privacy goal P2 (§2.3), Colo's protocol adds a zero-knowledge range proof [11]. Specifically, v_B commits to the values in T and proves that each value is bounded and that each value is offset by the same private mask r. Recall from the setup step (§4.1) that the keys for the ZKP are specific to length $len(T)$ of array T. Thus, v_B splits up proof generation as needed depending on the binary representation of $len(T)$. After generating the proofs, v_B sends them to v_A along with one entry $T'[j]$ of T' (using OT) as well as the opening for the commitment to $T'[j]$. Now, instead of using an OT protocol secure only against an honest-but-curious sender, Colo's protocol switches to an OT that defends against a malicious sender (v_B) [6, 26].

Colo's protocol is not general purpose and is not meant to replace Yao's GC. In particular, its overhead is proportional to the input domain size of the predicate F. Thus, it only works for small input domains. However, its performance benefits are very significant, and allow the devices to keep their overhead low. First, node v_B evaluates the predicate F in plaintext rather than inside Yao's GC. Second, the protocol uses ZKP for range proofs, a computation which has received significant attention in the literature [4, 7]. Third, and similar to the point above, the OT primitive is also a basic secure computation primitive that has received much attention on performance optimizations [6, 21, 26, 32].

4.3.2 Hiding topology. The local aggregation protocol so far hides node and edge data; however, an adversary that can monitor network traffic can infer who is running secure computation with whom and thus figure out the topology data for the devices. As noted earlier (§3), Colo protects this information by enabling the devices to communicate over a metadata-hiding communication system. In particular, Colo uses a state-of-the-art system called Karaoke [17].

Though Karaoke adds both significant CPU and network overhead to the servers (§6), we pick Karaoke for several reasons. First, the overhead is at the server-side, where there is more tolerance for overhead relative to the devices. Second, Karaoke can scale to millions of devices as needed for our scenario. Third, it defends against malicious servers who may duplicate or drop traffic to learn communication patterns. And, fourth, it provides a rigorous guarantee of (ϵ, δ) differential privacy.

4.4 Global aggregation

The global aggregation phase is the last phase in query execution. Recall that at the end of the local aggregation phase, each device has a local result, say y_i. For instance, y_i equals the output of secure computation $T'[j]$ in Figure 4 or the mask r added by a device that constructed the array T'. The goal of the global aggregation phase is to aggregate these outputs across devices to wrap up the execution of the query. For this aggregation, each device secret shares (using additive shares in the field $\mathbb{F}$) its output y_i and sends one share to each server. Each server then locally computes $\sum_i [y_i]$, where $[y_i]$ is a share it receives. The analyst finally aggregates the results across the servers, that is, computes $\sum_{edge}(T[j] + r) + \sum_{edge}(-r)$, canceling out the random masks, and obtaining the query output.

This simple global aggregation protocol ensures that the local aggregation results of an honest device get aggregated with the results of other honest devices exactly once. For instance, if a malicious server drops or duplicates a share $[y_i]$ supplied by an honest device, then the analyst $\mathcal{A}$ learns a uniformly random output, because the shares of $[y_i]$ at an honest server will not cancel out. This protocol guarantees privacy as long as one of the servers is honest, which is satisfied by our assumption of only up to $f = 20\%$ of servers being malicious (and the rest are honest).

4.5 Security analysis

Colo's technical report [20] includes an analysis of its privacy guarantees (P1 and P2). Briefly, a device's data is protected because of

the following reasons. First, the local aggregation secure computation with the help of ZKP and OT prevents an adversary from learning node or edge data (a neighbor cannot force another node to pick an arbitrary output for predicate F, and the node picks the predicate output obliviously while hiding its own input from the neighbor). Second, the use of metadata hiding communication network ensures that a network adversary cannot see traffic patterns. And, third, the use of secret shares ensures that honest devices' data is included exactly once in global aggregation, leading to bounded contribution in the query result.

5 IMPLEMENTATION

We have implemented a prototype of Colo in C++. Our prototype is approximately 2,000 lines of code on top of existing frameworks and libraries. The source code is available on Github.[1]

Parameters. We use Groth16 [11] as our underlying zkSNARK (ZKP) scheme and we use BLS12-381 as its underlying curve for a 128-bit security. For commitments, our prototype uses a ZKP-friendly scheme called Poseidon [10]. For OT, we use the simplestOT protocol [6]. For the metadata private messaging system, servers follow Karaoke's default configuration to generate noise messages to achieve $\epsilon = \ln 4, \delta = 10^{-4}$ differential privacy after 245 rounds of message losses.

6 EVALUATION

Our evaluation focuses on highlighting the device- and server-side overhead of Colo, for a variety of queries and we refer the readers to our technical report [20] for more experiments that vary device degree and the number of devices.

Baselines. We compare Colo to two baseline systems: a federated non-private baseline and the state-of-the-art Mycelium [31] system for privacy-preserving federated graph analytics.

We emphasize that Mycelium has a different and a stronger threat model than Colo—Mycelium assumes a single byzantine server, while Colo assumes a set of servers (e.g., 40) of which 20% can be byzantine. Thus, Mycelium is not a direct comparison (but the closest in the literature), and we use it to situate Colo's costs, as both Mycelium and Colo operate in strong threat models.

Testbed. Our testbed is Amazon EC2 and we use machines of type c5.4xlarge to run the devices and the servers. Each such machine has 16 cores, 32 GiB memory, and costs $0.328 per hour. For experiments, we don't run all devices, e.g., 1M, in one go, which would require thousands of machines. Instead, we sequentially run batches of 1K devices each.

Main results. Our evaluation compares the overhead of Colo and the baselines for different queries, that is, different sizes of input domain $len(T)$ of the query predicate. For these experiments, we fix the number of devices to 1M where each device has at most 50 neighbors.

Device-side overhead. Figure 5 shows per-device CPU and network overhead as a function of $len(T)$. Colo's overhead for $len(T) = 2$ (i.e., query Q1, Q2, Q4, and Q7 in Figure 1) is 10.27 s and 415 KiB, and increases to 8.42 min and 4.93 MiB for $len(T) = 256$ (this covers all queries in Figure 1), and 29.9 min and 18.12 MiB for $len(T) = 1000$.

Colo incurs much more overhead than the non-private baseline. For instance, for $len(T) = 256$, the baseline's overhead is 0.003 s and 6.36 KiB per device, while Colo's overhead is $1.68 \cdot 10^5$ and 775 times higher (505.1 s and 4.93 MiB) for the CPU and the network, respectively. However, relative to Mycelium, Colo's overhead is lower. For instance, for $len(T) = 256$, a device in Mycelium requires 31, 650 s CPU time (8.79 h) and 5.73 GiB network transfers, while Colo's 505.1 s and 4.93 MiB is 62.6× and $1.16 \cdot 10^3$× lower.

Colo's cost is dominated by local aggregation as the expensive metatdata-hiding communication is instantiated over the resourceful servers, and the global aggregation is also lightweight (§3, §4.3.2, §4.4). Further, the dominant local aggregation requires ZKP-friendly commitments and range proofs alongside OT, primitives that have been optimized in the literature (§4.3.1).

Although Colo's overheads are lower, they increase with $len(T)$ as the number of cryptographic operations in Colo's local aggregation protocol depend linearly on $len(T)$. For instance, the CPU time of a Colo device is dominated by the time to generate the ZKP—that each entry of T is within a range. This cost depends on the number of entries, with the proof for a single entry taking 0.04 s. Similarly, a Colo device's network overhead increases approximately linearly with $len(T)$; for instance, a device sends one commitment per entry of T. In contrast, the overheads for the baselines do not depend on $len(T)$ as their computation model multiplies $v_A.input \cdot v_B.input$ directly, rather than enumerating all possible outputs. Thus, Colo's overheads will surpass that of Mycelium for a large $len(T)$.

Overall, for Colo's target queries, its device-side overheads (in the range of a few seconds to a few minutes in CPU time, and a few hundred KiBs to a few MiBs in network transfers) appear to be in the realm of practicality for modern mobile devices.

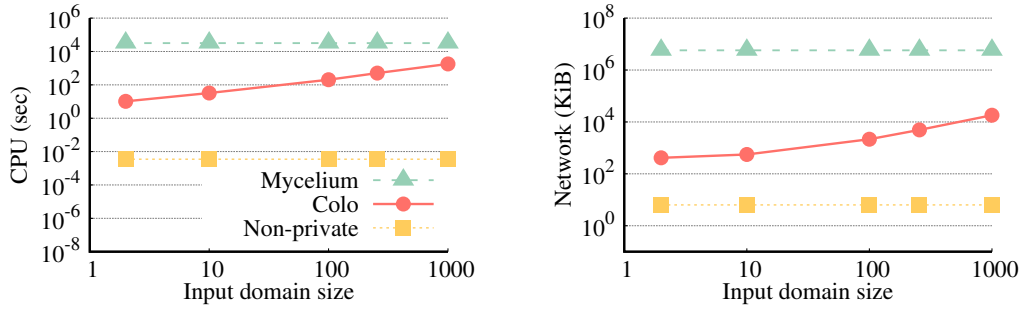
Server-side overhead. Figure 6 shows the server-side overhead for Colo and the baseline systems. Colo's server-side cost for $len(T) = 2$ is 88 h CPU time (on a single core) and 214.78 GiB per server, increases to 147.36 h and 3.46 TiB for $len(T) = 256$, and 224.17 h and 12.79 TiB for $len(T) = 1000$.

As with the device-side overhead, Colo's costs are significantly higher than the non-private baseline, whose server incurs 58.6 min CPU time and 6.36 GiB network transfers. Relative to Mycelium, Colo's server-side CPU is comparable, while the network is significantly lower. For instance, for $len(T) = 256$, the server-side cost for Mycelium is 1304 h and 5737 TiB, while it is 5894 h and 138.4 TiB for Colo's 40 servers combined.

Although Colo's server-side costs are substantial, since servers are resourceful (e.g., the CPU time can be split across many cores), we consider it affordable. We further assess this affordability by calculating the dollar cost of renting the servers.

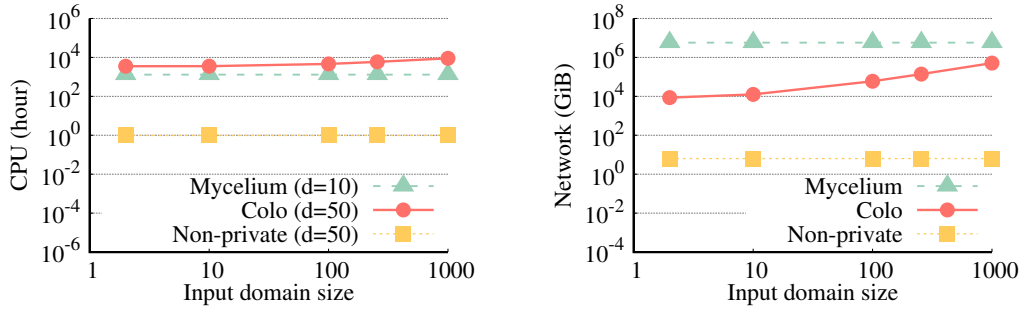
If applying a pricing model where the CPU cost per hour is $0.0205 and the cost of transferring one GiB is $0.01, Colo's dollar cost per query (over 1M devices) ranges from $3.95 to $37.6 per server, or $158 to $1504 total for the 40 servers, depending on the query. In contrast, the non-private baseline costs $0.08 and Mycelium costs $57,490 per query. These figures for Colo are substantial but within the reach of the budget of an entity like CDC. For instance, running the cheapest query (which includes the superspreader query) every two weeks will cost less than $4K annually.

[1] https://github.com/lonhuen/colo



(a) **Device CPU** (b) **Device network**

Figure 5: **Device-side cost with a varying input domain size** $len(T)$ **of the query predicate. Queries Q1, Q2, Q4, Q7 from Figure 1 have** $len(T) = 2$**, Q3, Q5, Q6 have** $len(T) \in [60, 240]$**, and Q8 has** $len(T) = 240$**.**



(a) **Server CPU** (b) **Server network**

Figure 6: **Total server-side cost for Colo (across its 40 servers combined) and the baselines with a varying input domain size** $len(T)$ **of the query predicate. The total number of devices is 1M and the degree of each is 50 (except for Mycelium's CPU cost where it is 10).**

7 RELATED WORK

Private data federation [1, 14, 22] focuses on a scenario where a set of data owners hold private relational data and a query coordinator orchestrates SQL queries on this data. The main difference with Colo is that these existing works target a scenario with a few data owners (e.g., a few tens) where each holds a significant partition of the relational data. In contrast, Colo targets millions of participants. While most works in private data federation consider parties to be honest-but-curious, it is natural in Colo to consider malicious parties given their number.

DStress [27] focuses on graph queries and a larger number of participants: a few thousand. The key use case is understanding systemic risk for financial institutions, which requires analyzing inter-dependencies across institutions. However, DStress also assumes honest-but-curious participants as financial institutions are heavily regulated and audited, and thus unlikely to be malicious.

Gunther et al. [13] consider malicious devices alongside a set of semi-honest servers to answer epidemiological queries such as an estimate of the change in the number infections if schools are closed for 14 days. However, they are limited to secure aggregation across neighbors' data and their protocol does not hide the result of local aggregation.

Mycelium [31] is the closest related work to Colo. It supports a broad set of queries, targets a large number of devices and assumes they can be malicious. However, as discussed earlier and evaluated empirically (§6), its costs are very high. Colo is a more affordable alternative in a strong threat model, at least for the subset of queries that Colo supports.

8 SUMMARY

Privacy-preserving federated graph analytics is an important problem as graphs are natural in many contexts. It specifically is appealing because it keeps raw data at the devices (without centralizing) and does not release any intermediate computation results except for the final query result. However, the state-of-the-art prior work for this problem is expensive, especially for the devices that have constrained resources. We presented Colo, a new system that operates in a strong threat model while considering malicious devices. Colo addresses the challenge of gaining on efficiency through a new secure computation protocol that allows devices to compute privately and efficiently with their neighbors while hiding node, edge, and topology data (§4.3.1, §4.3.2, §3). The per-device overhead in Colo is a few minutes of CPU and a few MiB in network transfers, while the server's overhead ranges from several dollars to tens of dollars per server, per query (§6). Our conclusion is that Colo brings privacy-preserving federated graph analytics into the realm of practicality for a certain class of queries. Future work included extending Colo to support richer queries and malicious integrity, and further lowering its overhead for query execution.

ACKNOWLEDGMENTS

The presentation of this paper was greatly improved by the careful comments of anonymous reviewers of PETS. This work is supported in part by a UCSB Academic Senate Grant, a UCSB Early Career Faculty Acceleration Award, and a National Science Foundation Award (2126327).

REFERENCES

[1] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. SMCQL: Secure querying for federated databases. *Proceedings of the VLDB Endowment*, 10(6), 2017.

[2] Q. Bi, Y. Wu, S. Mei, C. Ye, X. Zou, Z. Zhang, X. Liu, L. Wei, S. A. Truelove, T. Zhang, et al. Epidemiology and transmission of COVID-19 in 391 cases and 1286 of their close contacts in Shenzhen, China: A retrospective cohort study. *The Lancet infectious diseases*, 20(8), 2020.

[3] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *Journal of computer and system sciences*, 37(2), 1988.

[4] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy (S&P)*, pages 315–334, 2018.

[5] Centers for Disease Control and Prevention. https://www.cdc.gov/.

[6] T. Chou and C. Orlandi. The simplest protocol for oblivious transfer. In *International Conference on Cryptology and Information Security in Latin America*, 2015.

[7] H. Chung, K. Han, C. Ju, M. Kim, and J. H. Seo. Bulletproofs+: Shorter proofs for a privacy-enhanced distributed ledger. *IEEE Access*, 10:42067–42082, 2022.

[8] L. Danon, J. M. Read, T. A. House, M. C. Vernon, and M. J. Keeling. Social encounter networks: Characterizing Great Britain. *Proceedings of the Royal Society B: Biological Sciences*, 280(1765), 2013.

[9] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *ACM Symposium on Theory of Computing (STOC)*, 1985.

[10] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon: A new hash function for Zero-Knowledge proof systems. In *USENIX Security Symposium*, 2021.

[11] J. Groth. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 305–326, 2016.

[12] D. F. Gudbjartsson, A. Helgason, H. Jonsson, O. T. Magnusson, P. Melsted, G. L. Norddahl, J. Saemundsdottir, A. Sigurdsson, P. Sulem, A. B. Agustsdottir, et al. Spread of SARS-CoV-2 in the Icelandic population. *New England Journal of Medicine*, 382(24), 2020.

[13] D. Günther, M. Holz, B. Judkewitz, H. Möllering, B. Pinkas, T. Schneider, and A. Suresh. Privacy-preserving epidemiological modeling on mobile graphs. *arXiv preprint arXiv:2206.00539*, 2022.

[14] F. Han, L. Zhang, H. Feng, W. Liu, and X. Li. Scape: Scalable collaborative analytics system on private database with malicious security. In *IEEE International Conference on Data Engineering (ICDE)*, 2022.

[15] Q.-L. Jing, M.-J. Liu, Z.-B. Zhang, L.-Q. Fang, J. Yuan, A.-R. Zhang, N. E. Dean, L. Luo, M.-M. Ma, I. Longini, et al. Household secondary attack rate of COVID-19 and associated determinants in Guangzhou, China: A retrospective cohort study. *The Lancet Infectious Diseases*, 20(10), 2020.

[16] R. Laxminarayan, B. Wahl, S. R. Dudala, K. Gopal, C. Mohan B, S. Neelima, K. Jawahar Reddy, J. Radhakrishnan, and J. A. Lewnard. Epidemiology and transmission dynamics of COVID-19 in two Indian states. *Science*, 370(6517), 2020.

[17] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 711–725, 2018.

[18] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, volume 25, pages 680–722, 2007.

[19] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 2012.

[20] K. Liu and T. Gupta. Making Privacy-preserving Federated Graph Analytics with Strong Guarantees Practical (for Certain Queries). *arXiv preprint arXiv:2404.01619*, 2024.

[21] D. Mansy and P. Rindal. Endemic oblivious transfer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 309–326, 2019.

[22] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *ACM SIGMOD*, 2009.

[23] P. Mohassel and B. Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In *Advances in Cryptology—CRYPTO*, 2013.

[24] J. Mossong, N. Hens, M. Jit, P. Beutels, K. Auranen, R. Mikolajczyk, M. Massari, S. Salmaso, G. S. Tomba, J. Wallinga, et al. Social contacts and mixing patterns relevant to the spread of infectious diseases. *PLoS medicine*, 5(3), 2008.

[25] B. Nikolay, H. Salje, M. J. Hossain, A. D. Khan, H. M. Sazzad, M. Rahman, P. Daszak, U. Ströher, J. R. Pulliam, A. M. Kilpatrick, et al. Transmission of Nipah virus—14 years of investigations in Bangladesh. *New England Journal of Medicine*, 380(19), 2019.

[26] M. Orrù, E. Orsini, and P. Scholl. Actively secure 1-out-of-N OT extension with application to private set intersection. Cryptology ePrint Archive, Paper 2016/933, 2016.

[27] A. Papadimitriou, A. Narayan, and A. Haeberlen. Dstress: Efficient differentially private computations on distributed data. In *ACM European Conference on Computer Systems (EuroSys)*, 2017.

[28] Y. J. Park, Y. J. Choe, O. Park, S. Y. Park, Y.-M. Kim, J. Kim, S. Kweon, Y. Woo, J. Gwack, S. S. Kim, et al. Contact tracing during coronavirus disease outbreak, South Korea, 2020. *Emerging infectious diseases*, 26(10), 2020.

[29] M. O. Rabin. How to exchange secrets with oblivious transfer. *Cryptology ePrint Archive*, 2005.

[30] Recent Cyber Attacks & Data Breaches In 2023. https://purplesec.us/security-insights/data-breaches/.

[31] E. Roth, K. Newatia, Y. Ma, K. Zhong, S. Angel, and A. Haeberlen. Mycelium: Large-scale distributed graph queries with differential privacy. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[32] L. Roy. SoftSpokenOT: Communication-computation tradeoffs in OT extension. Cryptology ePrint Archive, Paper 2022/192, 2022.

[33] TraceTogether. https://www.tracetogether.gov.sg/.

[34] I. Vojinovic. Data breach statistics that will make you think twice before filling out an online form. https://dataprot.net/statistics/data-breach-statistics/.

[35] A. C. Yao. Protocols for secure computations. In *Annual Symposium on Foundations of Computer Science (SFCS)*, 1982.

[36] A. C. Yao. How to generate and exchange secrets. In *Annual Symposium on Foundations of Computer Science (SFCS)*, 1986.

[37] ZoKrates. https://zokrates.github.io/.