# Mycelium: Large-Scale Distributed Graph Queries with Differential Privacy

Edo Roth
University of Pennsylvania

Karan Newatia
University of Pennsylvania

Yiping Ma
University of Pennsylvania

Ke Zhong
University of Pennsylvania

Sebastian Angel
University of Pennsylvania
Microsoft Research

Andreas Haeberlen
University of Pennsylvania

**Abstract.** This paper introduces *Mycelium*, the first system to process differentially private queries over large graphs that are distributed across millions of user devices. Such graphs occur, for instance, when tracking the spread of diseases or malware. Today, the only practical way to query such graphs is to upload them to a central aggregator, which requires a great deal of trust from users and rules out certain types of studies entirely. With Mycelium, users' private data never leaves their personal devices unencrypted, and each user receives strong privacy guarantees. Mycelium does require the help of a central aggregator with access to a data center, but the aggregator merely facilitates the computation by providing bandwidth and computation power; it never learns the topology of the graph or the underlying data. Mycelium accomplishes this with a combination of homomorphic encryption, a verifiable secret redistribution scheme, and a mix network based on telescoping circuits. Our evaluation shows that Mycelium can answer a range of different questions from the medical literature with millions of devices.

## 1 Introduction

Personal devices collect massive amounts of data that can enable fascinating applications. For instance, the words typed by smartphone users could be (and in fact are) used to train predictive typing models, which allows phones to offer helpful word completions to users when they are typing. As another example, the data collected by contact-tracing applications (via Apple and Google's Exposure Notifications API) could be used to understand how diseases spread, or what environmental factors play a role. These are instances of *federated analytics* (FA), whereby users, each of whom has a device with some data, collaborate with an *aggregator* in order to answer questions such as "how often does the word 'system' appear after the word 'operating'?".

Of course, user data—including infection status and demographic information—is very sensitive. Without assurances on who will access their data or what insights will be drawn from it, many users will not comfortably participate in an FA system. One approach taken by prior work [17, 38, 42, 80, 81] is to design the system to provide *differential privacy* [34], a strong and mathematically rigorous privacy guarantee. With differential privacy, FA systems can safely aggregate information like the frequency of words from billions of user devices while preserving the privacy of individuals.

While privacy-preserving FA systems have made considerable progress [80, 81], existing systems lack support for *graph queries* such as: "if a device is infected with malware, how many of their contacts are infected within a week?". This is unfortunate, since graph queries can help study the spread of malware, disease, and misinformation; they could also test for "filter bubbles" and other social phenomena.

However, supporting graph queries privately is challenging due to fundamental differences from the queries traditionally studied in past FA work. In earlier systems, each device analyzes only its local data (e.g., the words that the local user has typed), and the answers are aggregated securely across devices. But in a graph query, each device needs information from *other* devices before it can provide its answer. For instance, in the above example, even though each user may know the identities of their contacts, they would need to find out which of their contacts have been infected. Such an operation raises three technical challenges:

- **Topology privacy**: How can vertices communicate with other vertices without leaking the sensitive topology of the graph to the aggregator? This is especially difficult when the only entity guaranteed to know how to reach all vertices is the aggregator itself (e.g., a user may know the IDs or names of their friends, but not their IP addresses).
- **Neighbor data privacy**: How can vertices collect data from their neighbors and use it to produce their own answer without violating the privacy of their neighbors? For instance, in the above example, how can we prevent users from learning their friends' infection status?


*SOSP '21, October 26–29, 2021, Virtual Event*

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483585

- **Scalability**: How can the system support queries across millions of devices? While it might be possible to build an FA that operates over graphs using secure multi-party computation across devices, these approaches do not scale.

To address these challenges, this paper introduces *Mycelium*, the first FA system to support queries on massive graphs distributed across a large number of participants. To address scalability, Mycelium's key insight is that, for many graph queries, we can divide the computation into two steps: (1) *local* computations that run in parallel on a small neighborhood of each vertex and output a vector of local results, and (2) a global aggregation step that combines the vertex-level results into a single global output. This is analogous to how frameworks such as Pregel [65] structure their queries, albeit for different reasons. Mycelium cannot support every Pregel query because not all of them are differentially private, but Mycelium's computation model is still quite general.

To guarantee topology privacy, Mycelium needs to provide a way for users' devices to communicate with each other so that they can obtain the inputs needed to execute their local computation (vertex program). This is difficult in many applications without disclosing the existence of the communication to the aggregator. For example, the COVID-19 exposure notification systems use pseudonyms for each device, and there is no obvious way to communicate with the owner of a pseudonym once it has moved out of Bluetooth range. Mycelium solves this problem by using the aggregator as a rendezvous point, while preventing it from learning the topology of the graph in the process. The key idea is a new mix network and a telescoping circuit mechanism inspired by Tor [31] that allows devices to forward their requests via other devices until the requests reach their destinations (§3). To guarantee neighbor data privacy, Mycelium uses homomorphic encryption to aggregate encrypted histograms that are sufficient to answer many queries of interest. We will show several examples of such queries in Figure 2.

A key challenge with Mycelium's mix network is that devices are unlikely to all be simultaneously online, so a fast mixing round could miss some devices—with consequences for both privacy and accuracy. To compensate, Mycelium uses long communication rounds (on the order of hours), so all devices have a chance to contribute their answer; the aggregator buffers messages as needed. Because of the long delays, Mycelium is not suitable for interactive queries; it is intended for longer-term social studies, such as disease spread, investment patterns, or information propagation.

We have implemented a prototype of Mycelium, and we use a combination of small-scale benchmarks and extrapolation to show that it can scale to millions of devices. The cost to the aggregator is well within the means of a typical data center, and the costs to individual devices are moderate: for a typical query, each device will incur around 430 MB of bandwidth and spend 15 minutes of computation. A small, randomly chosen set of devices will need to spend more, but the costs are comparable to what prior FA systems [80, 81] require at similar scales, even though these systems do not support graphs. In summary, our contributions are:

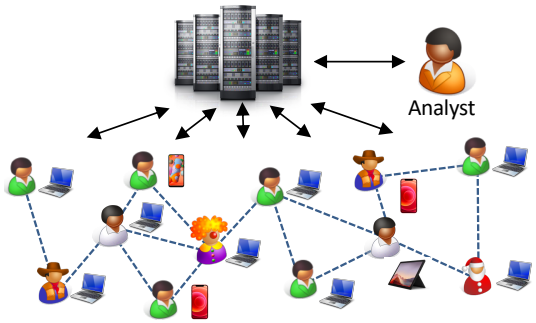



**Figure 1.** Millions of participants form a graph. An analyst submits queries to an aggregator who facilitates computing on the graph.

- A mix network with verifiable telescoping circuits (§3);
- Mycelium: the first FA system to support graphs (§4);
- A prototype implementation (§5) and experimental evaluation (§6) of Mycelium.

## 2 Federated analytics over graphs

We target a setting (illustrated in Figure 1) where there are a large number of *participants*, each of whom has a personal device that contains sensitive information (e.g., financial records, demographic information, health details). Each participant is identified by one or more *pseudonyms*, and participants may know some of the pseudonyms of other participants. For instance, in the case of Google and Apple's Exposure Notification System (GAEN) [2], the devices are users' smartphones; the sensitive information includes users' infection status, time of diagnosis, and locations visited; the pseudonyms could be the Rolling Proximity Identifiers (RPIs), which each phone broadcasts to other nearby phones via Bluetooth Low Energy, or some fixed identifier. Overall, we can think of this data as representing a large graph, with one vertex for each participant and a directed edge $(p_1, p_2)$ whenever $p_1$ knows at least one of $p_2$'s pseudonyms.

There is also a central *aggregator*, who wishes to run large-scale queries over this graph and is willing to coordinate the necessary computation. Note that these queries are not *real-time* queries; at this scale, they may take hours or days to complete. We assume that the aggregator has substantial computational and bandwidth resources, perhaps in the form of a data center. The aggregator works with at least one *analyst*, who formulates the queries to be run. In the case of GAEN, the aggregator could be Google or Apple, or the government agencies that run the Diagnosis Servers; the analysts could be some carefully vetted epidemiologists.

We assume that devices are usually (though not always) online. Devices could be behind NATs or firewalls, or they

| Query | Application | Description |
|---|---|---|
| Q1 | [6, 78] | Histogram of the number of infections in an infected participant's two-hop neighborhood, within 14 days<br>`SELECT HISTO(COUNT(*)) FROM neigh(2) WHERE dest.inf ∧ self.inf` |
| Q2 | [28, 68, 73] | Histogram of the amount of time $A$ has spent near $B$, if $A$ is infected within 5-15 days of contact with $B$<br>`SELECT HISTO(SUM(edge.duration)) FROM neigh(1) WHERE self.inf ∧ (dest.tInf∈[edge.last_contact+5,edge.last_contact+10])` |
| Q3 | [16, 28, 68] | Histogram of the frequency of contact between $A$ and $B$, if $A$ infected $B$<br>`SELECT HISTO(SUM(edge.contacts)) FROM neigh(1) WHERE self.inf ∧ dest.tInf ∧ (dest.tInf>self.tInf+2)` |
| Q4 | [16] | Secondary attack rate of infected participants if they travelled on the subway<br>`SELECT HISTO(SUM(dest.inf)) FROM neigh(1) WHERE onSubway(edge.location) ∧ self.inf` |
| Q5 | [68] | Histogram of the number of distinct contacts within the last 24 hours, for different age groups<br>`SELECT HISTO(COUNT(*)) FROM neigh(1) GROUP BY self.age` |
| Q6 | [28, 52, 68] | Histogram of secondary infections caused by infected participants in different age groups<br>`SELECT HISTO(COUNT(*)) FROM neigh(1) WHERE self.inf ∧ dest.tInf ∧ (dest.tInf>self.tInf+2) GROUP BY self.age` |
| Q7 | [16, 48, 68] | Histogram of secondary infections based on type of exposure (such as family, social, work)<br>`SELECT HISTO(COUNT(*)) FROM neigh(1) WHERE self.inf ∧ dest.tInf ∧ (dest.tInf>self.tInf+2) GROUP BY edge.setting` |
| Q8 | [52, 75] | Secondary attack rates in household vs non-household contacts<br>`SELECT GSUM(SUM(dest.inf)/COUNT(*)) FROM neigh(1) WHERE self.inf GROUP BY isHousehold(edge.location)` |
| Q9 | [58, 68] | Secondary attack rates within case-contact pairs in the same age group vs different age groups<br>`SELECT GSUM(SUM(dest.inf)/COUNT(*)) FROM neigh(1) WHERE dest.age∈[0,100] ∧ self.age∈[dest.age-10,dest.age+10]` |
| Q10 | [52] | Secondary attack rates at different stages of the disease (incubation period vs illness period)<br>`SELECT GSUM(SUM(dest.inf)/COUNT(*)) FROM neigh(1) WHERE self.inf ∧ (dest.tInf>self.tInf+2) GROUP BY stage(dest.tInf-self.tInf)` |

**Figure 2.** Example queries. `CLIP` commands and histogram bins have been omitted.

could go offline for brief periods of time due to loss of cellular coverage or whenever they run out of power.

### 2.1 Example queries

We now provide a few examples of queries that we wish to support. For concreteness, we focus on queries proposed in the infectious disease literature, even though Mycelium is general and can handle graph queries for other domains. Figure 2 summarizes the queries, along with the motivating works, and the corresponding SQL-like syntax.

Superspreading is a well-established phenomenon for infectious diseases [37, 62], and there is work that quantifies the role of superspreaders in pandemics [6, 16, 58, 61]. For example, two works [6, 78] investigate data containing information about chains of transmission or clusters originating from a primary source. Such queries can be formulated as which calculate the number of infected individuals in the N-hop neighborhood of the primary source.

Another line of research analyzes the conditions under which infections most likely occur [16, 28, 48, 52, 58, 68, 75]. In particular, these works calculate secondary attack rates (the probability that an infected individual transmits the disease to an exposed contact) [52, 58] under various conditions. For example, several works [16, 28, 48, 52, 58, 68, 75] explore secondary attack rates of infected individuals across sex, age, household sizes, and epidemic phases; others [16, 48, 68] explore secondary infections based on exposure type. User devices provide access to location and demographic data, which makes such queries possible. Additionally, with temporal data we can answer queries such as Q2 and Q3.

Right now, these queries are answered through manual tracing; for instance, one study uses data from 391 cases and 1,286 of their close contacts in China [16]. A deployment in a GAEN-like system could potentially provide access to larger data sets. Even in cases where data is collected by a country's public health system [75], privacy concerns still exist [33]. A system like Mycelium would allow queries over sensitive data without violating the privacy of individuals.

Although these queries look different, they are structurally similar: they (1) look at a small "neighborhood" around each vertex in the graph, such as the vertices within two hops; (2) compute something across this neighborhood, such as the number of infections; and finally (3) compute some aggregate statistic about these numbers, such as a histogram.

### 2.2 Threat model and goals

We assume that all parties—the participants, the aggregator, and the analysts—could be potentially malicious (Byzantine). However, following prior work [80, 81], we use the *OB+MC assumption*: we assume (1) the aggregator is honest-but-curious at the beginning and usually thereafter, but could be occasionally Byzantine (OB) for brief periods, and (2) most of the participants are correct (MC), except for perhaps 1–2%. OB basically models a system compromise or an inside attacker who may control the aggregator arbitrarily, but only for a short period of time. If the aggregator were malicious all the time, it could manufacture an unbounded number of colluding Sybils, defeating all known defenses. With 100 million devices, MC still means that there will be 2 *million* Byzantine participants. Our goal is to provide the following properties:

- **Output privacy:** The output of the query should not leak (much) information about the data of individual users, or about the presence or absence of particular edges.

- **Neighbor data privacy:** The computation that is used to answer the query should not reveal anything about a given user's sensitive data to other users.
- **Topology privacy:** The computation should not reveal the presence or absence of an edge to the aggregator.

Notice that we do *not* try to achieve topology privacy between users; our solution does leak a very small amount of information about the topology to nearby users, which is the presence of multiple paths between two users. This is out of necessity: if we tried to perfectly hide the topology even from nearby users, we could not avoid double-counting data from different pseudonyms of the same user, which would severely limit accuracy. However, users already know, or can know, most of the information that is being leaked, since edges are formed through formal relationships or physical proximity. Another non-goal is that we do *not* try to protect the aggregator from itself: if the aggregator tells lies or otherwise misbehaves during one of its Byzantine periods, it can permanently lose the ability to ask additional queries and would then have to reinitialize the entire system.

In addition to the above three properties, we are interested in solutions that can efficiently scale to millions of participants and do not require additional trusted parties.

### 2.3 Background: Differential privacy

For output privacy, we adopt *differential privacy* [34], a formal definition that bounds how much an adversary can learn about an individual participant from the output of (randomized) queries over a database – in our case, the graph. Informally, a query is differentially private if adding or removing one participant's data results in "almost no change" in the probability distribution of the output. This guarantee is quantified with a parameter, $\epsilon$, that controls how much the distribution over the output can vary. Formally, a query $q$ is *$\epsilon$-differentially private* if, for any graphs $d_1$ and $d_2$ that differ in one vertex and the edges connected to that vertex, and any set of outputs $R$, $\Pr[q(d_1) \in R] \leq e^{\epsilon} \cdot \Pr[q(d_2) \in R]$. That is, removing one participant results in at most a multiplicative change of $e^{\epsilon}$ in the probability of any set of outputs.

A standard method for achieving differential privacy for numeric queries is the *Laplace mechanism* [34], which involves two steps: first calculating the *sensitivity* $s$ of the query—which is how much the un-noised output can change based on removing a single user—and second, adding noise drawn from a Laplace distribution with scale parameter $s/\epsilon$; this results in $\epsilon$-differential privacy.

In general, differential privacy is difficult to achieve for graph data because graph properties are highly sensitive to changes in vertices and edges. For instance, an undirected linear graph with $n$ vertices has diameter $n$, but the addition of a single edge between the first and the last vertex cuts the diameter to $\frac{n}{2}$. However, the queries in Table 2 are fairly local; they basically count the vertices whose $k$-hop neighborhood has a certain property. This type of query tends to have a low sensitivity bound that can be computed statically (§4.7).

### 2.4 Strawman solutions

To illustrate the challenges of this scenario, we discuss two strawman solutions.

**Plain text.** Participants could upload their data and the observed pseudonyms to the aggregator, who could answer queries with standard systems such as GraphX[45] or GraphLab [63]. However, this requires users to trust the aggregator, since it can learn the data and the edges of all users.

**MPC.** Multi-party computation (MPC) [87] is a way for multiple parties to jointly compute a function on their private data, such that no party learns anything beyond what the output of the function implies. A large MPC between *all* participants that aggregates results and adds noise could achieve our privacy goals, but we are not aware of any MPC that can scale beyond a few hundred participants, whereas our scenario can involve millions.

### 2.5 Our approach

Our key insight is that scalability can be achieved by splitting the computation into two parts: a local part that can be executed by the devices themselves, by exchanging messages with other devices that they share an edge with, and a global part that efficiently aggregates the results of the local part. We adapt Orchard [81] for the global aggregation (§4.2); Mycelium's key contributions are the local computation for graphs, the communication mechanism between devices, and eliminating the need to generate new cryptographic keys for each query. (At the scale of millions of devices, key distribution is a significant source of overhead and complexity.)

Mycelium executes queries as *vertex programs*, analogous to queries in Pregel [65]. Each vertex has some local state, which is initially the private data of the corresponding participant. The computation then proceeds in discrete rounds that each consist of a *communication step* and a *computation step*. In the communication step, each vertex can send a message to each of its direct neighbors in the graph; in the computation step, each vertex can optionally update its state, based on the messages it has received. After a fixed number of rounds, each on the order of an hour, each vertex must set its state to a vector of numbers. These vectors are then summed up in a final *aggregation step*, which also adds the noise that is required for differential privacy and then outputs the final vector of noisy sums.

The separation into a local and a global part is key to scalability because it preserves the information about the graph topology. Recall from Section 2.1 that queries in our scenario typically examine a small local area around each vertex (e.g., the two-hop neighborhood). Thus, the data of each vertex can influence at most a small, constant number of other vertices. If $d$ is an upper bound on this number,

and $N$ is the number of devices, we can compute the final result with $O(N \cdot d)$ operations. But if the topology of the graph is encrypted, the information about *which* vertices can influence each other is lost; any vertex could potentially influence any other vertex. Thus, there is no obvious way to avoid operations on all possible pairs of vertices, resulting in $O(N^2)$ operations. With millions of vertices, this can make a difference of several orders of magnitude.

# 3 Communication

In Mycelium's local phase, the devices need to be able to exchange messages with their direct neighbors in the graph, without giving away details of the topology. This is not completely straightforward, because (a) the devices only know their neighbors' pseudonyms, not their identities or IP addresses, and (b) since the devices can be behind firewalls and occasionally go offline, a device and its neighbor may not be able to establish a direct connection, or may never even be online at the same time. We solve this problem using a type of mix network where devices act as mixes and the aggregator acts as an (untrusted) mediator for all messages.

## 3.1 Assumptions and goals

Our goal is a primitive $\textsc{send}((h_1, m_1), \ldots, (h_d, m_d))$ that delivers a set of messages $\{m_1, \ldots, m_d\}$ to the holders of pseudonyms $\{h_1, \ldots, h_d\}$, respectively, with high probability. We make the following assumptions:

1. There is an upper bound $d$ on the degree of each vertex.
2. Devices' clocks are loosely synchronized.
3. Devices have a key pair $(pk_i, sk_i)$ for each pseudonym $h_i$, and $pk_i$ is linked to the pseudonym $h_i$ ($h_i = H(pk_i)$).
4. All devices know (a) a tight upper bound, $N_D$, on the number of devices, and (b) a bound $P$ on the number of pseudonyms that a valid device could have generated within the time period for which a query is valid.
5. There is a public bulletin board (blockchain) that prevents the aggregator from equivocating to the devices.

## 3.2 High-level approach

At a high level, we use onion routing. A device $s$ sends a message $m$ to a pseudonym $t$, by routing $m$ through a chain of $k$ other devices: $s$ chooses $k$ pseudonyms $h_1, \ldots, h_k$ and then sends $Enc_{sk_1}(Enc_{sk_2}(\ldots, (Enc_{sk_k}(Enc_{sk_t}(m)))))$ to the first hop $h_1$; $h_1$ removes a layer of encryption and sends the result to $h_2$; and so on, until $h_k$ sends the message $m$ to the destination $t$. If $k > 1$ and at least one device on the chain is honest, the edge between $s$ and $t$ is hidden.

Since devices cannot communicate directly, Mycelium relays messages through the aggregator, who maintains a "mailbox" per pseudonym. This must be done with care: if devices pick up the messages from their mailboxes one at a time, the aggregator could observe that a message deposited by Alice is picked up by Bob, and that Bob then deposits a message in Charlie's mailbox—revealing the chain. We address this by proceeding in discrete rounds and ensuring that each device mixes and forwards different messages in each round. (We call these C-rounds to distinguish them from rounds of the vertex program.) Thus, the aggregator can only observe, say, that Bob picks up several messages, including Alice's, and that Bob then deposits messages in several other mailboxes, including Charlie's. If each device forwards a batch of $b$ messages in each C-round, and there are at most $c$ devices on the chain colluding with the aggregator, then a given message could be in $b^{k-c}$ mailboxes after $k$ C-rounds. For sufficiently large $b$ and $k$, and small $c$ in expectation, this makes it hard for an adversary to link messages.

The above presupposes that devices are always online, that colluding devices are honest but curious, and that the aggregator does not drop messages. We discuss how to handle a malicious aggregator later. To handle devices that go offline or drop messages, it is not sufficient to send a single copy of a message to a target as the message may never reach it. To guard against this, each device sends $r$ replicas of each message over different chains. Additionally, to hide the vertex degrees, each device always sends $d$ different messages; if it has fewer than $d$ neighbors in the graph, it sends extra messages to itself, somewhat analogous to Loopix [77].

If every device sends messages to $d$ targets and uses $r$ replicas of each message, the expected batch size is $b = r \cdot d$. Since bigger batches lead to better security, we restrict the choice of hops to a random fraction $f$ of the nodes. This means that when a device is selected as a routing node, it will handle more messages but be selected less frequently. This increases the batch size by a factor $1/f$, without increasing the average workload.

## 3.3 Initialization

To make the above approach work, devices must be able to pick random pseudonyms for building their chains, without giving the aggregator a way to bias the choice towards colluding devices. For this purpose, the aggregator creates a verifiable map $M_1$ that maps each integer in $[1, N_D \cdot P]$ to a different pseudonym. Since a malicious aggregator could omit pseudonyms or include pseudonyms more than once, it is required to also create a second map $M_2$ that can be used to audit the first map. This works as follows.

When a new query is issued, the aggregator begins by compiling a list of the $P$ most recent pseudonyms each device has used. It then randomly assigns each device a unique *device number* in the range $[1, N_D]$, and each pseudonym a unique *pseudonym number* in the range $[1, N_D \cdot P]$. Next, it creates $M_1$ as a binary Merkle hash tree (MHT), whose leaves are of the form $(h_i, pk_i, d_i)$, where $h_i$ is the $i$-th pseudonym, $pk_i$ is the corresponding public key, and $d_i$ is number of the device that owns the pseudonym. To ensure that the devices have a consistent view, the aggregator then commits to $M_1$ by posting the root of the MHT to a bulletin board.

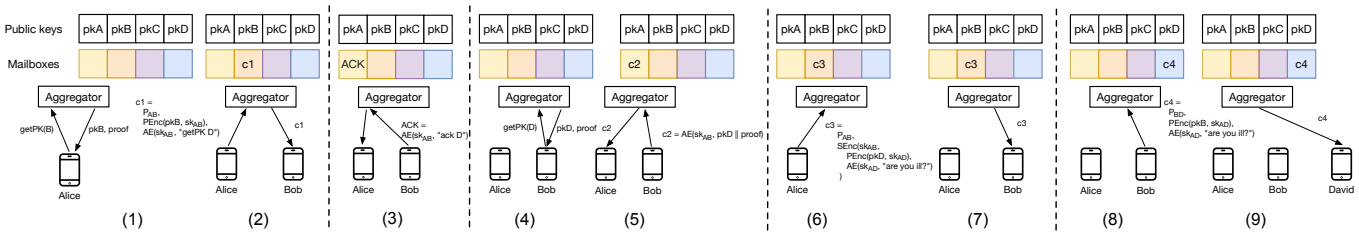



**Figure 3.** Steps for relaying a first message from Alice to David through Bob (so $k = 1$). Rounds are separated by vertical lines. Steps (1)–(5) correspond to the first 3 rounds and are for path establishment; Steps (6)–(9) are for forwarding. Alice gets the key for pseudonym B (Bob's), directly from the aggregator in Step 1 and then asks B to look up the key for D (David's) in Step 2. After Bob sends an ACK to A (Step 3), it waits $k$ rounds, looks up the key for D, and sends it to A (Steps 4–5). Finally, Alice sends her message along the path (Steps 6–9). `PEnc` is public key encryption, `SEnc` is symmetric encryption, and `AE` is authenticated encryption as discussed in Section 3.5.

Using this information, a device could theoretically look up the $n$-th pseudonym and its public key by sending $n$ to the aggregator. The aggregator could then take a binary representation of $n$ and walk down $M_1$'s MHT starting from the root, taking a left on level $i$ if the $i$-th bit of $n$ is zero, and a right otherwise. This would take it to the $n$-th vertex from the left. The aggregator could then return that vertex's information to the device, along with an inclusion proof (hashes along the path from the leaf to the root), and the device could verify the response by checking that (a) the pseudonym matches the public key, and (b) the path in the inclusion proof matches the path the aggregator should have taken for $n$. In practice, such a direct lookup would tell the aggregator that the device is using the $n$-th pseudonym in a chain; we discuss how to fix this in Section 3.4.

To enable the devices to audit $M_1$, the aggregator also prepares another verifiable map $M_2$, which maps each device number to a leaf $(H(h_1), \ldots, H(h_P), H(pk_1), \ldots, H(pk_P))$, where the $h_i$ are the pseudonyms this device has used and the $pk_i$ are the device's public keys. The root of this tree is posted to the bulletin board as well. Each device then performs two checks using $M_1$ and $M_2$. First, it looks up its *own* pseudonyms in $M_1$ and checks the inclusion proofs. Thus, if the aggregator has omitted an honest device's pseudonyms, that device will detect the problem. Second, each device randomly looks up $x$ pseudonyms, extracts the corresponding device numbers $d_i$, and asks the aggregator to show that one of the $H(pk_j)$ hashes in the $d_i$-th leaf of $M_2$ corresponds to the pseudonym the device has retrieved. If a device submits a lot more than $P$ pseudonyms, this check will fail with high probability, since each of $M_2$'s leaves can hold only $P$ entries; if a device assumes multiple identities, the aggregator will run out of space in $M_2$, which can have only $N_D$ leaves.

Starting with the posting of the MHT roots, devices use their clocks to mark the fixed length of each C-round.

### 3.4 Path setup

Each device randomly selects $r$ $k$-hop "paths" for each of the $d$ messages it will send. Recall that the hops should be picked from among a fraction $f$ of the devices. The devices select each hop $i$ from 1 to $k$ by picking a random number $x$ from $[1, N_D \cdot P]$ such that $(i-1) \cdot f \leq \frac{H(x \,||\, B)}{H_{max}} < i \cdot f$, where $H$ is a cryptographic hash function, $H_{max}$ is the maximum hash value, and $B$ is a random bitstring that is chosen collectively as, e.g., in Honeycrisp [80]. Notice that at this point the position of each pseudonym in $M_1$ is fixed, so a malicious adversary cannot bias the selection towards its confederates.

So far, the devices know only the index of their desired hops in $M_1$. However, they need to know the actual pseudonyms and establish a shared (symmetric) key with each hop. They cannot ask the aggregator for the pseudonyms directly, since this would give away the intended path. Instead, we use a variant of the telescoping scheme from Tor [31], which we describe next (and illustrate in Figure 3). For ease of exposition, we discuss the protocol in terms of a single device and a single path $h_1, \ldots, h_k$, but the steps are done in parallel across all devices' $d \cdot r$ paths.

In the first C-round, the source device $s$ looks up the pseudonym $h_1$ and public key $pk_1$ for its first hop by communicating directly with the aggregator. (In this step and all that follow, the response includes both the leaf and the inclusion proof, and the device verifies them in the same way as in Section 3.3.) This is safe because the aggregator will be able to observe the connection to the first hop anyway. $s$ then generates a symmetric key $sk_{s \leftrightarrow h_1}$ and a random path id $p_{s \leftrightarrow h_1}$, and uses an authenticated encryption scheme (AE) to encrypt the identity of the next hop, $h_2$, with $sk_{s \leftrightarrow h_1}$; $s$ encrypts $sk_{s \leftrightarrow h_1}$ using public key encryption ($PEnc$). Finally, $s$ deposits $p_{s \leftrightarrow h_1} || PEnc(pk_1, sk_{s \leftrightarrow h_1}) || AE(sk_{s \leftrightarrow h_1}, h_2)$ in $h_1$'s mailbox.

Once all devices have deposited their messages, the aggregator computes (a) a *mailbox MHT* over the messages in each mailbox, and then (b) a *C-round MHT* over all the inner MHTs. It then commits the root of the C-round MHT in the bulletin board, and then proves to each sender that its messages were included in the MHT. This prevents the aggregator from dropping messages without detection. If some devices do not receive the proof from the aggregator, they post a challenge on the bulletin board. If the aggregator did receive messages from these devices, it can respond with the correct proofs; if a challenge is not answered, the

other devices refuse to proceed, and the path setup has to be restarted without the relevant devices.

Next, $h_1$ retrieves the batch of messages from its mailbox, and asks the aggregator to reveal the MHT for this mailbox so that it can verify that it has received all the messages. If no misbehavior is detected, $h_1$ looks up all the public keys for the requested pseudonyms (e.g., $h_2$) in a random order. Then, $h_1$ deposits in $s$'s mailbox $AE(sk_{s\leftrightarrow h_1}, pk_2)$, which corresponds to $h_2$'s public key encrypted under the shared symmetric key between $s$ and $h_1$. At the end of the C-Round, $s$ checks its own mailbox and decrypts the message to learn $h_2$'s public key $pk_2$.

During the next C-Round, $s$ generates a fresh symmetric key for $h_2$, $sk_{s\leftrightarrow h_2}$, encrypts it under $pk_2$, and sends to $h_1$: $p_{s\leftrightarrow h_1}||AE(sk_{s\leftrightarrow h_1}, PEnc(pk_2, sk_{s\leftrightarrow h_2})||AE(sk_{s\leftrightarrow h_2}, h_3))$. $h_1$ then fetches messages from its mailbox and checks the MHT. Finally, $h_1$ removes the outer layer of encryption of the message from $s$, generates a new path id $p_{h_1\leftrightarrow h_2}$, locally stores the map $p_{h_1\leftrightarrow h_2}$ to $p_{s\leftrightarrow h_1}$ (to be used in later rounds), and deposits $p_{h_1\leftrightarrow h_2}||PEnc(pk_2, sk_{s\leftrightarrow h_2})||AE(sk_{s\leftrightarrow h_2}, h_3)$ in $h_2$'s mailbox.

This process continues hop by hop: $h_2$ looks up the key for $h_3$, $h_3$ the key for $h_4$, and so on, until $h_k$ is finally asked to look up the key for the destination *dst*. One issue is that when $h_k$ receives a batch of requests to fetch the public keys of the final destinations (one of which is *dst*), $h_k$ cannot proceed right away. This is because a malicious penultimate hop ($h_{k-1}$) could drop the final request sent by $s$ where it tells $h_k$ to fetch *dst*'s key. If $h_k$ were to fetch the keys immediately, the aggregator would observe that *dst*'s public key is fetched fewer times than every other device's, thereby revealing that a device who had $h_{k-1}$ as a penultimate hop had an edge to *dst*—shrinking the anonymity set of *dst*'s edge from $(r \cdot d/f)^k$ to $(r \cdot d/f)^{k-1}$ possible sources. To avoid shrinking the anonymity set, $h_k$ sends an ACK to all sources through the reverse paths confirming that it has received their requests (if a source doesn't get an ACK it complains). If $h_k$ does not see any complaint in the bulletin board in $k$ rounds (the number of C-rounds it takes ACKs to get back to sources), $h_k$ goes ahead and fetches the public keys in its batch (including *dst*'s). If a source complains, then the last hops in all paths refuse to fetch public keys, and path setup is restarted. On any reverse path to the source, each honest hop knows the number of messages it should receive, and also refuses to proceed if any message is dropped.

At the end, each device knows the pseudonyms and public keys for all the hops along its chosen paths, and it has established a shared symmetric key with each hop. With $k$ hops, this process requires $2+4+6+\ldots+2k+k = k^2+2k$ C-rounds. However, $k$ should normally be small, and the process is run infrequently in order to let new devices join the system. With $k = 3$ and one-hour C-rounds, path setup would take about half a day, which gives flexibility to devices so they can participate even if they briefly go offline.

### 3.5 Message forwarding

Once the paths are set up, communication is as follows. One communication round of the vertex program requires $k+1$ C-rounds. In the first C-round, the devices onion-encrypt their messages, as described in Section 3.2, and deposit them in the mailboxes of their first hops with the path ids generated during the path setup. Then, in each subsequent C-round, each hop downloads the messages from their mailbox, checks to make sure the aggregator did not drop any messages, removes one encryption layer, and mixes them. Finally, they use the mapping generated during path setup to determine the appropriate path id for each message, and upload these with each message to the mailbox of the next hop.

A complication is that the failure of a device could give away some message paths to the aggregator. For instance, suppose that, in a previous round, Charlie downloaded messages from Alice and Bob, and uploaded messages to Doris and Eliot, but in the current round, the message from Alice is missing. If Charlie were to upload a message only to Eliot, the aggregator would be able to conclude that Doris was the next hop after Alice on some path. To counteract this, each hop uploads a dummy message for the next hop for which they do not have a valid message. That way, the communication pattern remains unchanged.

**Generating dummies.** If messages between a source $s$ and each hop $h_i$ in its path are encrypted with authenticated encryption (AE), then it is infeasible for a forwarding device to generate dummies that decrypt properly owing to AE's existential unforgeability guarantee. This enables the following attack. Let the attacker control $h_{i-1}$ and $h_{i+1}$. The attacker uses $h_{i-1}$ to drop a message, so $h_i$ generates a random dummy to mask the missing message, and then the attacker uses $h_{i+1}$ to detect which of the messages it received are invalid—thereby learning the relationship between the input and output path ids of $h_i$. To prevent this, we observe that we only need ciphertext integrity between $s$ and the destination *dst*. Hence, $s$ can construct an onion encryption where the inner ciphertext (the message to *dst*) uses an AE that is indistinguishable from a random message of the same length. For example, encrypt with a stream cipher, then MAC with a PRF, and use the monotonically increasing round number as a nonce (not sent with the ciphertext to avoid known privacy pitfalls [14]). All other onion layers use a symmetric cipher that is indistinguishable from random but that lacks a MAC. This allows $h_i$ to generate a random dummy message which $h_{i+1}$ cannot detect as invalid.

## 4 Query processing

Mycelium evaluates queries in two stages: first, each vertex evaluates a *local* query over its own $k$-hop neighborhood (say, to compute the number of infected contacts this vertex has), and then the results of the local queries are summed up, noised, and reported to the analyst (say, as a histogram

showing what fraction of users have a certain number of infected contacts). In the following, we will refer to the vertex at the "center" of a given local query as the *origin vertex*. Mycelium uses a subset of SQL, with two small extensions, to specify the local queries.

Conceptually, the queries "see" the data as a table `neigh(k)` that contains a row for each member of the $k$-hop neighborhood, including the origin vertex. The columns of this table are: (a) the private data of the origin vertex (`self`); (b) the private data of the relevant neighbor (`dest`); and (c) the private data associated with the first edge on the path from the origin to the neighbor (`edge`). Queries can ask for `COUNT`s and `SUM`s over columns; we obviously cannot allow direct queries for private data. The `WHERE` predicate can use conjunctions and disjunctions, as well as arbitrary tests within the same column group (e.g., a comparison of two `self` values). It can also contain inequalities over values from different column groups (e.g., `dest.tInf>self.tInf+2`, as in Q3, to test whether a neighbor was diagnosed more than 2 days after the origin vertex) *as long as* both take a finite number of discrete values. Finally, queries can use `GROUP BY` over `self` columns to report statistics for different attribute values.

One extension to SQL is that queries must choose whether the outputs of the local queries should simply be summed up globally (`GSUM`), perhaps to compute a secondary attack rate as in Q8, or aggregated into a histogram (`HISTO`), as in Q1. Another extension is that `GSUM` queries must specify a "clipping range" $[a, b]$; if the computed value is below or above this range, it is clipped to $a$ or $b$, respectively.

### 4.1 HE encoding

The two biggest challenges with our protocol are (1) how to implement histograms, and (2) clipping without compromising output privacy or neighbor data privacy.

Suppose, for instance, that we wanted to compute how many users have between 0 and 2, between 3 and 5, and more than 5 infected contacts. Naïvely, we would use private comparisons to implement this: each contact encrypts either 0 or 1, depending on whether they are infected, and sends the ciphertext to the origin vertex, which computes the sum `S` of the values and then uses homomorphic encryption (HE) to compute, say, `IF (0<=S<=2) THEN 1 ELSE 0` for the first bin of the histogram. However, private comparisons between ciphertexts and plaintexts are *extremely* expensive.

Instead, we use the following technique. We rely on the *leveled homomorphic* cryptosystem[1] by Brakerski-Gentry-Vaikuntanathan (BGV) [20], whose plaintexts are polynomials of degree $N$ with integer coefficients, and we encode the value $a$ (e.g., 0 or 1 in the above example) as the polynomial $x^a$. Then we can use BGV's homomorphic multiplication to add up encoded values: if a device receives $Enc(x^a)$ and $Enc(x^b)$ from two neighbors, it can compute $Enc(x^{a+b}) = Enc(x^a) \cdot Enc(x^b)$. BGV's homomorphic addition then becomes a "bin" aggregation: if one receives $Enc(x^0+x^1)$ and $Enc(x^0 + x^2)$, then summing these ciphertexts produces $Enc(2x^0 + x^1 + x^2)$, which is an encrypted polynomial where the $i$-th coefficient gives the number of times that bin $i$ was selected. We can also compute the values in a coarser bin, say $[0, 2]$, by adding up the coefficients of $x^0$, $x^1$, and $x^2$.

The price to pay is that (1) our encoding cannot support more bins than the degree $N$ of the polynomial, (2) the number of local summands cannot exceed the number of multiplications BGV can support, and (3) the number of values to be aggregated cannot exceed the range of the coefficients. This seems fine in our setting: we use $N = 32,768$, which is far larger than, say, the number of infected friends a given user can have; for reasonable parameters, BGV can support dozens of multiplications; and, with a plaintext modulus of $2^{30}$, we can "bin"-aggregate more than a billion values.

### 4.2 Aggregation with Orchard

Orchard [81] has the ability to answer a range of non-graph queries, in an otherwise similar setting to ours. The workflow of Orchard also requires a homomorphic encryption scheme, albeit only a simpler *additive* one. Devices encrypt their data and send them to a central aggregator, who sums up ciphertexts. However, the aggregator does not hold the keys for decryption—instead, they are secret shared among a randomly elected *committee* of 10–20 user devices, which use MPC to perform key generation and decryption. The aggregator first uses a summation tree to prove to each device that its data has been included in the sum exactly once; then it sends the aggregate ciphertext to the committee, which decrypts it, adds noise for differential privacy, and returns it back to the aggregator as the final result to the query. This process can be composed over multiple queries, as long as a privacy budget is tracked (see Section 4.4).

Mycelium makes two modifications to Orchard. First, it replaces Orchard's additively homomorphic cryptosystem with BGV [20], in order to support both homomorphic additions and multiplications. Second Mycelium observes that, in prior FA systems (including Orchard), each time an analyst wants to run a new query the system must generate and distribute new cryptographic keys to all devices. For systems with millions or billions of devices, such key distribution is both costly and complex. Instead, Mycelium leverages a *verifiable secret redistribution* scheme (VSR) [46] to generate all the cryptographic keys *once*, distribute them to all devices, and then transfer the corresponding private key from one committee to another in such a way that members of different committees cannot collude to recover the key.

In more detail, at the beginning of Mycelium's operation, a set of non-colluding parties, which we call the *genesis committee*, generates all the necessary public keys (including *relinearization keys* which the BGV scheme uses to keep ciphertext small after multiplications) and keep secret shares of

[1] A leveled HE supports additions and a small number of multiplications.

the corresponding decryption key such that no non-majority of parties can reconstruct the decryption key.

The genesis committee will then transfer ownership of the decryption key shares to the first randomly chosen committee in Mycelium using VSR. Subsequent rounds of Mycelium will likewise perform a VSR transfer of the decryption key from the old committee to a new committee, completely eliminating the need for Orchard's expensive key generation phase. We give more details in Section 5.

### 4.3 Basic protocol: Single hop

We first give a protocol where a vertex can answer a query that requires information about its immediate neighbors, and then generalize to a $k$-hop neighborhood in Section 4.4. Processing a query `SUM` over a particular attribute such as `SUM(dest.inf)` consists of the following steps. First, the origin vertex sends a query ID $q$ to all of its neighbors, so they know to which query to respond. Second, each neighbor sends back to the origin vertex a ciphertext $Enc(x^b)$. In the case of a `SUM`, $b$ is the value of the attribute; in the case of a `COUNT`, $b$ is 1 if the predicate applies, and 0 otherwise. After collecting the ciphertexts from each of the neighboring vertices, the origin vertex sums up the encoded values by multiplying the received ciphertexts together, as discussed in Section 4.1. The result is a ciphertext of the form $Enc(x^i)$, where $i$ represents the result of the local query over the origin vertex's local neighborhood.

As we discuss later, all of these ciphertexts are then globally aggregated using BGV's additive homomorphism, resulting in a final ciphertext of the form $Enc(\sum_{i=0}^{N-1} c_i x^i)$, where $c_i$ is the number of origin vertices that obtained $i$ as the result of their local query.

### 4.4 Basic protocol: Multiple hops

We now generalize the above protocol to $k$-hop neighborhoods. For now we assume queries that do not (1) use `GROUP BY`, (2) compute sums over edges, or (3) compare fields from different column groups. In Table 2, Q1, Q2, and Q4 are of this type. For simplicity, we will assume that the `WHERE` predicate is already in conjunctive normal form.

**Flooding.** A query over the $k$-hop neighborhood `neigh(k)` proceeds in $2k$ rounds. As in the single-hop case, in the first round each origin vertex sends a query ID $q$ to its neighbors. In the following $k-1$ rounds, these messages flood to the $k$-hop neighborhood as follows. When a node receives a message with a given query ID, it remembers from which neighbor it got it. We call this neighbor the *upstream neighbor*. The message from the upstream neighbor is forwarded to all other neighbors. Thus, at the end of the $k$-th round, each node in the $k$-hop neighborhood of each origin vertex (a) has received a message from that vertex, (b) knows its upstream neighbor, and (c) knows its distance from the origin vertex, which is simply the number of the round in which the message with a given query ID was first received.

**Processing:** In the $k+1$-th round, for each upstream neighbor, each vertex evaluates the arguments of each `SUM` or `COUNT` over its local data; for instance, if the query asks for a `SUM(dest.inf)`, each node would look up its infection status, yielding a local result $r_i$. Next, the vertex evaluates the `dest` clauses of the `WHERE` predicate over its local data; if they all evaluate to `true`, the vertex computes $Enc(x^{r_i})$. If one of the predicates evaluates to `false`, it computes $Enc(x^0)$. Finally, each vertex at distance $k$ from the origin takes each encrypted result and then SENDS it to the relevant upstream neighbor. If a node drops off in the middle of a computation, their value defaults to $Enc(x^0)$, and will thus have a neutral effect on the query's results. From a privacy perspective, this leaks no information about the node's underlying data.

**Local aggregation.** In round $k-i$, each vertex at distance $k-i$ from the origin receives a ciphertext from each of its neighbors. The vertex evaluates the `dest` clauses and, if they all evaluate to `true`, it multiplies all ciphertexts together, along with an encryption of its own value. The effect is that the vertex now holds an encryption of the *sum* of the encoded values that have been aggregated so far. Finally, unless the vertex is the origin vertex, it sends the result to its upstream neighbor. If a clause evaluates to `false`, it sends $Enc(x^0)$.

**Final processing.** In round $2k$, the origin vertex holds a ciphertext which contains the aggregated values over the entire $k$-hop neighborhood. The origin vertex then evaluates the `self` predicates from the `WHERE` clause; if any evaluate to `false`, it replaces the ciphertext with $Enc(0)$. The origin vertex then contributes the ciphertext for global aggregation.

**Global aggregation.** The global aggregator receives the ciphertexts from all of the origin vertices and sums them all up. Then, the aggregator gives these ciphertexts to the committee who has the corresponding decryption key (§4.2). The committee then decrypts the final ciphertext and adds a calibrated amount of noise based on the query before releasing the result. In particular, let $p$ be the plaintext encoding the underlying aggregated values. For histogram queries, the coefficients of $p$ that fall into each bin of the histogram are summed up, and then, after adding some noise to each bin, the results are released to the analyst. For `GSUM` queries with a clipping range $[a, b]$, the committee clips the range of outputs by computing $\sum_{i=a+1}^{b-1} i \cdot p_i + a \cdot (\sum_{i=0}^{a} p_i) + b \cdot (\sum_{i=b}^{N} p_i)$ and then adds noise and releases the sum to the analyst.

**Privacy budget.** To bound the privacy loss from multiple queries, the committee maintains a "privacy budget" from which the $\epsilon$ cost of each new query is deducted. This is a common approach [34, §3] used in prior FA systems. Our prototype subtracts the full $\epsilon$ of each query from the budget, which is safe but conservative. There are several more sophisticated techniques, such as advanced composition theorems [36, §3.5] or sparse-vector techniques [80], that would stretch the budget further and that can be used instead.

### 4.5 Special cases

We now discuss how Mycelium handles the special cases excluded in Section 4.3. If a query contains a `GROUP BY`, the origin vertex does not just report a single value, but rather one for each possible combination of values in the grouped columns. Our homomorphic cryptosystem is designed such that all of these values can be packed into a single ciphertext. Only one of these—the one that corresponds to the origin vertex's values in the grouped columns—will represent a non-zero value; the others will be $Enc(0)$. For instance, for Q6, a 20-year old will report a value of 0 for all categories outside of the 18–25 category. Because the parameters of Mycelium support large ciphertexts (§5), it can support a fairly large range of possible values in the grouped columns.

If a query compares fields from `self` and `dest` columns, Mycelium does the following. Suppose the comparison is a clause `self.x>dest.y`, and the predicate also contains a `BETWEEN` clause that limits the values of column `y` to a discrete range $[a, b]$. Then, rather than sending back a single ciphertext $Enc(x^m)$, where $m$ is the value in the `y` column, the destination vertex reports a *sequence* of ciphertexts, one for each value in $[a, b]$, with $Enc(x^m)$ in the position corresponding to $m$, and $Enc(1)$ in all other positions.

During final processing, the origin vertex sums up the subsequence of size $\ell$ that corresponds to values greater than the value of `self.x`, and then subtracts $Enc(\ell - 1)$ from the sum. This means that the final summed value will be $Enc(1)$ if the destination vertex reported no value (or one outside of the subsequence). Otherwise, the final value will be exactly $Enc(x^m)$. This allows for correct multiplication with the other neighbors' ciphertexts. For example, for a subsequence of length 3, if the neighbor sent $Enc(1)$, $Enc(x^m)$, and $Enc(1)$, the origin vertex will add the ciphertexts received from the neighbor to get $Enc(2 + x^m)$, and then subtract $Enc(3 - 1) = Enc(2)$ from it to get $Enc(2 + x^m) - Enc(2) = Enc(x^m)$.

### 4.6 Malicious nodes

The above protocol returns the correct result if all of the nodes in a device's $k$-hop neighborhood are correct. But what if some of them are Byzantine? A Byzantine node may not follow the protocol and instead return ciphertexts with coefficients larger than 1, or with more than one nonzero coefficient; the result could be that the aggregator receives a value larger than $B$. Even if a device itself is correct, it cannot prevent this because it cannot tell what it is computing.

We use zero-knowledge proofs (ZKP) [44] to prevent this attack. When a node sends a ciphertext to its parent, we say that the ciphertext is *well-formed* if it is computed as described above. Each node sends a ZKP to prove to the aggregator that its ciphertext is well-formed. Additionally, each origin vertex sends a ZKP to the aggregator proving that it computed the local aggregation of its $k$ hop neighborhood correctly by multiplying the ciphertexts from its neighbors.

If the ZKP requires a trusted setup (such as Groth16 [47], which we use in our prototype), this setup is performed by the genesis committee (§4.2). There are also alternatives that do not require a trusted setup called *transparent zkSNARKs*.

### 4.7 Security analysis

**Output privacy.** By construction, all queries in our language have bounded sensitivity, and this bound can be statically determined by multiplying the maximum value contribution of any one device by the total number of devices in their local neighborhood. For `GSUM` terms, the max contribution is simply the size of the clipping range; for `HISTO` terms, it is always two because, by changing its local contribution, a vertex can at most decrease the count in one bin by 1 and increase the count in another, also by 1. Thus, we can simply use the Laplace mechanism to achieve differential privacy.

**Neighbor data privacy.** The message flow is independent of a vertex's private data—in the aggregation phase, each vertex sends back $Enc(0)$ if the `WHERE` predicate evaluates to false—and all the values are encrypted with HE, under a key that neither the aggregator nor individual nodes know.

**Topology privacy.** The flooding phase reveals to each node (a) the size of its $k$-hop neighborhood (which is equal to the number of distinct query IDs that arrive), and (b) the number of other node(s) within a $k$-hop radius that can be reached over more than one directly adjacent edge, and if so, over which edges (because in that case the same query ID arrives over each of these edges). Other than that, the nodes learn nothing about the topology: they only communicate with their direct neighbors, and the values in the messages they receive have already been aggregated by the neighbors.

**Malicious nodes.** If a given $k$-hop neighborhood contains some malicious nodes, these nodes can report incorrect partial sums for their own subtrees of the spanning tree, by encrypting any plausible value (from within $0..B \cdot (d+1)^\ell$, where $\ell$ is their level in the tree) and computing a matching ZKP, or simply by refusing to send a message to their parent in the tree. However, they cannot cause the aggregator to accept a vector with more than one non-zero coefficient or a vector where the value of the non-zero coefficient is greater than 1 because the aggregator verifies these properties using the ZKP and discards data from nodes whose ZKP is invalid. Thus, a small number of malicious devices cannot have a disproportionate impact on the overall result. We note that discarding invalid inputs introduces a bias towards the data from correct nodes, but (a) the effect should be small, due to the MC assumption, and (b) it seems hard to avoid since there is no way to tell what the *correct input* of a malicious client would have been.

**Traffic analysis.** Some mixnets, such as Tor, are vulnerable to traffic analysis attacks such as intersection and disclosure attacks [7, 27, 79], in which the adversary observes the traffic in the entire network over some time frame and then

makes inferences about whether or not certain participants are communicating. These attacks leverage the fact that mix networks are often *sparse*—that is, only a fraction of participants communicate in any one stage of the protocol. In Mycelium, *every* device participates in *every* mixnet stage, which renders these types of passive attacks infeasible.

### 4.8 Limitations

One obvious limitation of our approach is that there are useful queries that cannot be expressed in our query language. This is not a fundamental limitation—with HE and our communication mechanism from Section 3, it should be possible to execute *any* Pregel-like query, as long as the HE scheme supports enough multiplications and the cost of the additional communication rounds is acceptable. The key question is how one would prove differential privacy. Perhaps a query language such as Fuzz [49] or Duet [70], or even manual privacy proofs using apRHL [8] or CertiPriv [13] could help.

## 5 Implementation

For our prototype, we used Orchard's codebase [81] with three changes: we (1) replaced Orchard's HE scheme with BGV [20], which required reimplementing the MPC for decryption; (2) implemented a mechanism for adding Laplace noise in the MPC for decryption; and (3) replaced the ZKPs in Orchard with those of Section 4.6. We implemented our mix network from Section 3 in C++ using OpenSSL [3] for basic operations (e.g., encryption and decryption). We instantiated `PEnc` using RSA-PKCS1 public key encryption, `SEnc` using ChaCha20, and `AE` using ChaCha20-Poly1305 (nonce is not included in the message). For redistribution of the secret key (§4.3), we implemented Extended VSR [46].

**Security parameters.** For BGV, we set the plaintext modulus to $2^{30}$, the ciphertext modulus to a 550-bit prime, and polynomial degree $N$ to 32768. This set of parameters gives over 128 bits of security [9]) and supports 1-hop queries on over a billion users by encoding values of up to 30 bits.

To reduce computation costs on devices, we defer the relinearization for each multiplication to the global aggregation phase, where the aggregator performs a one-time operation to reduce ciphertext size before the decryption step.

**MPC and secret sharing.** We implemented MPC operations using version 1.7 of SCALE-MAMBA [53], which provides security against up to $\lfloor \frac{k-1}{2} \rfloor$ malicious parties, and performs operations in a finite field modulo a configurable prime $p$, which helps us support BGV decryption. SCALE-MAMBA also supports Shamir secret sharing [82]; we share the secret key among the $k$ committee members such that any subset of $t+1$ members can reconstruct the secret key, where $t \geq \frac{k}{2}$. At the same time, no $t'$ (where $t' \leq k/2$) dishonest nodes can learn anything about the key, and $t+1$ honest nodes can detect any errors introduced by dishonest nodes. Using the initial setup by the genesis committee (§4.2), the secret key is distributed to the first committee. Every committee then uses the extended VSR protocol [46] to generate new shares of the secret key for the subsequent committee.

| | | |
|---|---|---|
| Number of devices | $N$ | $1.1 \cdot 10^9$ |
| Onion routing hops | $k$ | 3 |
| Replicas of each message | $r$ | 2 |
| Fraction of forwarders | $f$ | 0.1 |
| Committee size | $C$ | 10 |
| Degree bound | $d$ | 10 |

**Figure 4.** The parameters we used, unless noted otherwise.

**Zero-knowledge proofs.** We use ZoKrates [4], a high-level language that can be consumed by SNARK compilers to produce circuits, to express our zkSNARK statements. These in turn can be used with many proof systems, some of which do not need a trusted setup. We use `bellman` [1] as the proof system, which implements the Groth16 scheme [47]. We implemented the proofs for encryption and ciphertext multiplication using this toolbox, and benchmarked the costs for proof size, proving time, and verification time.

## 6 Evaluation

This section addresses four questions: (1) How many queries can Mycelium support? (2) what are the major costs, to normal users, to committee members, and to the aggregator?, (3) how well does the onion routing protect topology privacy?, and (4) how well does Mycelium scale?

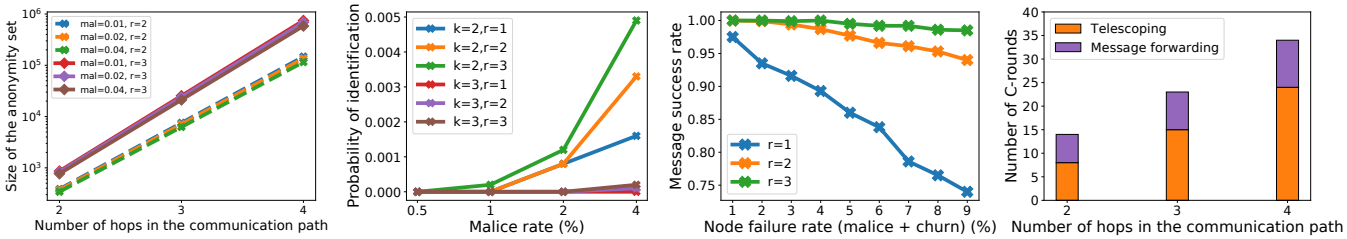
### 6.1 Experimental setup

Since we were not able to deploy a system with millions of nodes, we benchmark the various components separately, and extrapolate the costs at scale as done in Orchard [81]. For the client-side and aggregator-side HE benchmarks, we use a MacBook Pro with a 2 GHz quad-core processor and 16 GB of RAM. For our mix net, we run experiments on CloudLab [32] m510 machines with 8-core 2GHz processors and 64 GB of RAM; for the MPC benchmarks, we use 15 Amazon EC2 t2.xlarge instances with 16 GB of RAM. Figure 4 summarizes the parameters we use, unless specified otherwise.

### 6.2 Generality

We first examined the range of queries Mycelium can support. There are two reasons why Mycelium might not support a given query: (1) it is not expressible in the query language from Section 4, or (2) the HE scheme in our prototype may not be able to run enough multiplications to process it.

We tried to implement and run each of the queries in Table 2. All queries were expressible, and the query expression is included in the table. This is not surprising because the queries we found in the medical literature compute simple statistics, such as the number of patients for which a particular predicate is true. We were able to run all the queries




(a) Size of the anonymity set (b) Probability of identification (c) Goodput (d) Duration

**Figure 5.** Performance of Mycelium's communication layer

except Q1. The latter is a two-hop query that would require $d^2 = 100$ multiplications, which exceeds the noise budget of the HE scheme we chose. This is not an inherent limitation; recent HE libraries [5] are close to supporting this number.

This result suggests that Mycelium can already support many practical queries, which seems encouraging.

### 6.3 Communication layer

Next, we looked at the performance of Mycelium's anonymous communication layer. Recall that Mycelium onion-routes each message on $r$ different $k$-hop paths, and that, at each hop, each message is mixed with $(r \cdot d)/f$ messages. The aggregator can observe (a) the sets of encrypted messages each forwarder downloads and uploads, and (b) anything that the colluding forwarders saw.

We first focus on topology privacy. Suppose the adversary wants to learn whether there is an edge $(a, b)$. It can observe which messages $b$ downloads at the end, so it can reason about the set of senders that each message could have come from. Each honest forwarder increases the size of this set by $r/f$—the uploaded message could have been in any of the messages the same forwarder downloaded earlier. Thus, with $k$ honest hops, the number of possible senders is roughly $(r/f)^k$. However, the $r$ replicas of a given message would have come from the *same* sender, so in some cases, the adversary can intersect the $r$ sets. However, because there are more total messages in the system, and the probability of multiple intercepted messages is relatively low, increasing $r$ still (on expectation) leads to larger anonymity sets. Figure 5(a) shows how the expected set size changes with $r$ and $k$. For our parameters of $r = 2$ and $k = 3$, a malicious fraction of 0.02 still yields an anonymity set of over 7000 devices.

However, a node can be "unlucky" and choose a path that consists only of malicious nodes. In this case, the adversary can identify this exact node as the sender of the message. Figure 5(b) shows the probability for this case. With our default setting of $k = 3$, each query gives the adversary a chance of $p \approx 10^{-4}$ to identify a given edge.

Another concern is that message might not reach its destination because all $r$ copies are dropped—either on purpose, by malicious forwarders, or by accident if a forwarder goes offline and does not return by the end of the C-round.

| Queries | Number of ciphertexts |
|---|---|
| Q1, Q2, Q4, Q5, Q8 | 1 |
| Q3, Q6, Q7, Q10 | 14 |
| Q9 | 10 |

**Figure 6.** Number of ciphertexts sent for each query

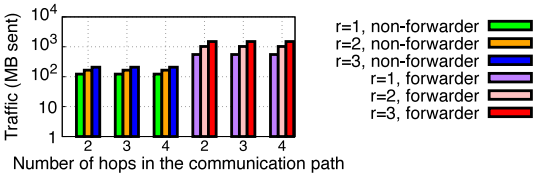



**Figure 7.** Avg. bandwidth required of each participant per query

Figure 5(c) shows "goodput," the probability that a given message is successfully received (without modifications by adversaries). With $r = 2$ and a node failure rate of 4% (including both malicious nodes and departures), only about one in 100 messages is lost completely. Queries can handle this case, e.g., by specifying a default value for missing inputs in the local aggregation, by counting the number of local aggregations where this (detectable) condition occurs, and/or by asking the local aggregators to upload a final value only when all inputs have been received.

A final question is how long forwarding takes. Figure 5(d) shows the number of C-rounds that are needed for telescoping ($k^2+2k$) and forwarding ($2k+2$, since each query requires a message for the query and a message for the response). If $k = 3$ and C-rounds are one hour long, then both phases of a one-hop query will finish in less than a day. (The duration depends only on the number of hops and not on what specifically the query computes.) This is fine, since Mycelium is not for real-time queries.

### 6.4 What is the cost for normal users?

Next, we examined the bandwidth and computation cost of Mycelium for normal user devices. Each device performs up to three operations: (1) it prepares its own contributions to its neighbors' local aggregations; (2) it potentially acts as a forwarder during onion routing; and (3) it completes a local aggregation for its own neighborhood.

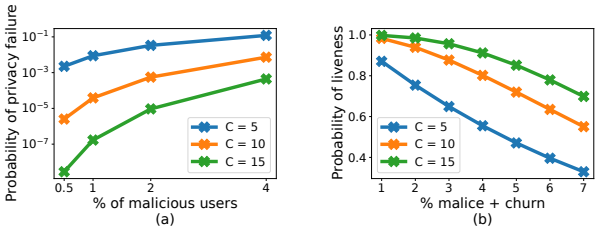



**Figure 8.** Probability of privacy failure (a) and liveness (b) with different committee sizes.

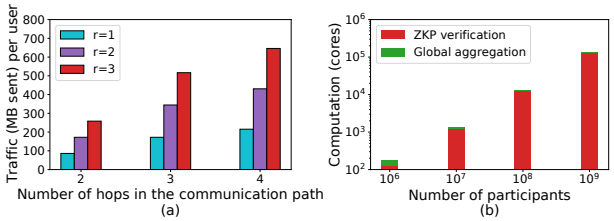



**Figure 9.** Per-user bandwidth (a) and total computation (b) required of the aggregator for each query.

The communication costs vary between queries, depending on how many FHE ciphertexts they require; each one is around 4.3 MB. Figure 6 shows the number ($C_N$) of ciphertexts for each of the queries in Table 2. In the following, we focus on the cost of a basic query with $C_N = 1$ ciphertext, such as Q5; for the more complex queries, the communication costs need to be multiplied by the number of ciphertexts.

Figure 7 shows the communication cost per device. The figure contains two column families: one for the case where the device is selected as a forwarder, and one for the case where it is not. For each case, we vary the number $k$ of hops during onion routing and the number $r$ of copies that are sent of each message. The costs are dominated by message forwarding: each device has to send $r \cdot C_N \cdot d$ large FHE ciphertexts, where $C_N$ is the factor from Figure 6, and, when chosen as a forwarder with probability $f$, it has to download and upload $(r \cdot C_N \cdot d)/f$ of these ciphertexts. For our default parameters from Figure 4 and a simple query with $C_N = 1$, this works out to 1030 MB for forwarders and 170 MB for non-forwarders, or around 430 MB on expectation, given that a $k \cdot f$ proportion of participants will serve as forwarders. For comparison, this is about the cost of sending a four-minute video attachment from an iPhone.

The computation time per device mainly depends on the time to perform ciphertext operations, including encryption and ciphertext multiplication for neighborhood aggregation, as well as the time to generate the ZKPs. The ciphertext operations take around 14 minutes in total with our Python implementation, and the ZKP proof generation takes around a minute, so the total computation time per device is roughly 15 minutes. We implemented an (unoptimized) version of BGV in Python for compatibility with the MPC and ZKP software, so these costs could be dramatically reduced to make use of existing HE optimizations. The computation times for telescoping and message forwarding were negligible, and the costs did not vary much between different queries.

### 6.5 What is the cost for committee members?

For each query, a small committee of $C$ user devices is expected to participate in the decryption MPC using their shares of the secret key. Our EC2 benchmarks show that, although Mycelium uses a different cryptosystem, the cost of this MPC is comparable to Orchard's: with a committee of size 10, the total computation time needed was around 3 minutes and the bandwidth required per member is around 4.5GB, plus the (negligible) bandwidth for resharing the secret key. With millions of devices, an individual user's chances of having to serve on a ten-member committee are very small; nevertheless, due to the high bandwidth, it may be best to rely on desktops or laptops where possible.

Figure 8 allows us to reason about the tradeoffs associated with using different committee sizes: a higher committee size provides more security over time (because a larger committee is less likely to contain a majority of malicious members), but also increases the bandwidth and computation time required. We made these graphs using equations obtained from the Honeycrisp authors. Figure 8(a) shows the probability of there being enough malicious members in the committee to reconstruct the secret key, thus causing a privacy failure. In this case, a new secret key must be constructed using a new trusted setup. Figure 8(b) shows the probability of enough committee members being present to perform decryption. If there aren't enough members for liveness, we simply have to wait for some amount of time before enough members are back, and retry the computation.

### 6.6 What are the costs to the aggregator?

Recall that all messages are sent through the aggregator, who maintains mailboxes for each device. Figure 9(a) shows the total amount of traffic the aggregator would need to send to each device, depending on the number of onion-routing hops $k$ and number of replicas per message $r$. As expected, there is a substantial amount: for our choice of $k = 3$ and $r = 2$, the aggregator would need about 350 MB per device, or roughly the size of a 10-minute 1080p YouTube video.

The aggregator also needs to verify the ZKPs of each user and perform a global aggregation of ciphertexts. Figure 9(b) shows the number of cores needed to finish the computation within 10 hours with different system sizes. The cost is dominated by the ZKP verification (the bars for the aggregation are very small). Although zkSNARKs normally have small, constant proof sizes, the scheme we use (Groth16) scales linearly in the public I/O size, which, in our case, includes the fairly large ciphertexts. If necessary, the aggregator could reduce this cost by spot-checking only a fraction of the ZKPs, or it could stretch the computation over a longer time.

## 7 Discussion

**Cost:** It is clear that Mycelium's privacy comes at a high cost—queries on non-sensitive data could be answered cheaply by simply uploading the data to the aggregator in the clear and using a traditional graph-processing system such as GraphX [45]. Indeed, we implemented Q1 for a 1-hop neighborhood in GraphX and ran it on a CloudLab machine with a random billion-node graph and random data. The query finished in about 5 seconds. Mycelium is meant for queries on highly sensitive data that would make the aggregator a target for attacks if it were collected in the clear, and queries that cannot even be asked today because no single aggregator can be trusted with the necessary data.

**Device heterogeneity:** In a practical deployment, one challenge would be the wide range of device capabilities. Serving as a communication hop or committee member seems fine for a laptop or workstation that is connected to a wired network, but could be problematic for a mobile phone with a metered cellular connection and limited battery capacity. However, we note that mobile devices are increasingly part of device federations (e.g., a laptop, mobile phone, and smartwatch all sharing the same iCloud account).Since the devices in a federation are typically owned by the same individual, they could safely share their data and designate the most powerful device—say, the laptop—as a participant in Mycelium.

Communication steps could also be delayed when a device is on the road, and resumed when it is plugged in and on a WiFi connection. Finally, hops and committee member selection could be biased towards more powerful devices; this would give the adversary a small advantage, since all of its confederates could claim to be powerful, but one could use slightly more aggressive parameter settings to compensate.

**Aggregator workload:** For the aggregator, the major costs are communication bandwidth and ZKP verification. Much of the bandwidth is due to the very large HE ciphertexts (4.3 MB), but we speculate that future HE schemes will eventually reduce this cost. For ZKP verification, we note that the 10-hour limit for Figure 9(b) was somewhat arbitrary; in practice, ZKP verification could be done in the background, whenever a data center has spare capacity, as long as the query results are not needed immediately.

## 8 Related Work

**Private analytics.** There is considerable work on differential privacy [35], some of which considers aggregating data from multiple domains [41]. However, most target relational data: PDDP [23] builds histograms and DJoin [69] computes database joins. Neither is sufficient to answer graph-based queries. DStress [74] can handle graph data, but does not scale beyond thousands of users. Of the systems that work at scale, including academic works [42, 80, 81], and deployed solutions [11, 15, 17, 19, 25, 30, 38, 39, 76], none handle graphs.

**Traditional graph processing.** Graph analytic frameworks [24, 29, 43, 45, 51, 57, 64, 71, 72, 86, 91] target scale but not privacy. Work on social networks has dealt with issues of anonymity [12, 40, 89, 90], but the proposed mechanisms either focus on answering limited differentially private queries [18], on aggregate network estimations that may hide effects of individual malicious nodes [50], or on previous definitions of privacy like k-anonymity [60].

**Private contact tracing.** Work in contact tracing does not support a single aggregator, or is not designed for central analytics. Mazloom and Gordon [66] support Pregel-like graph queries but require two servers to split trust between them, and does not guarantee differential privacy. Poirot [88] gives differentially private contact summary aggregation, but also splits trust amongst multiple servers, which perform a joint MPC. In the last year we have also seen the design of several other exposure notification and proximity detection systems that give user-level insights [2, 21, 83]. These insights include notifying individuals when they are likely to have been exposed to an infection, but do not support graph analytics.

**Anonymous messaging.** Mycelium's messaging layer is inspired by Tor [31]. However, Mycelium must operate without the equivalent of Tor relays, and since the devices themselves cannot necessarily communicate directly with each other, it has no choice but to relay communication through the aggregator, which is a global, active adversary. Mycelium's messaging can be seen as a different mix network architecture [10, 22, 26, 54–56, 59, 67, 77, 84, 85] that has high latency and prioritizes privacy over availability, but that has the benefit of not requiring prior pairwise sharing of cryptographic material between senders and the chosen mixes, and balances the load across different sets of mixes every run of the protocol, which helps the system scale to billions of users.

## 9 Conclusion

Mycelium is the first system to support differentially private analytics on graph queries at a massive scale. It leverages HE, a new mix-network, and Pregel-style queries on top of a Honeycrisp-like architecture. Because ciphertexts must support aggregation of up to a billion devices' information, the costs of Mycelium are higher than similar FA systems. Future work may incorporate cryptographic advances that improve these costs while supporting richer graph queries.

## Acknowledgments

We thank our shepherd Amit Levy and the anonymous reviewers for their thoughtful comments and suggestions. This work was supported in part by NSF grants CNS-1513694, CNS-1563873, CNS-1733794, CNS-1703936, CNS-1955670, CNS-2045861, CNS-2107147, and CNS-2124184; DARPA contract HR0011-17-C0047; a Google Faculty Research Award; and a JP Morgan Chase & Co Faculty Award.

## References


[1] bellman. https://github.com/zkcrypto/bellman.

[2] Exposure notifications: Using technology to help public health authorities fight COVID-19. https://www.google.com/covid19/exposurenotifications/.

[3] OpenSSL. https://www.openssl.org.

[4] ZoKrates. https://github.com/Zokrates/ZoKrates.

[5] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, 2020.

[6] D. Adam, P. Wu, J. Wong, E. Lau, T. Tsang, S. Cauchemez, G. Leung, and B. Cowling. Clustering and superspreading potential of severe acute respiratory syndrome coronavirus 2 (sars-cov-2) infections in hong kong. *Nature Medicine*, 2020.

[7] D. Agrawal and D. Kesdogan. Measuring anonymity: The disclosure attack. *IEEE Security & Privacy*, 1(6), Nov. 2003.

[8] A. Albarghouthi and J. Hsu. Synthesizing coupling proofs of differential privacy. *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.

[9] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. Homomorphic encryption security standard. Technical report, 2018.

[10] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[11] Apple Differential Privacy Team. Learning with privacy at scale. *Apple Machine Learning Journal*, 2017.

[12] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography. In *International World Wide Web Conference (WWW)*, 2007.

[13] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. In *Proc. POPL*, 2013.

[14] M. Bellare, R. Ng, and B. Tackmann. Nonces are noticed: AEAD revisited. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2019.

[15] A. Bhowmick, J. Duchi, J. Freudiger, G. Kapoor, and R. Rogers. Protection against reconstruction and its applications in private federated learning. *arXiv:1812.00984 [cs, stat]*, 2018.

[16] Q. Bi, Y. Wu, S. Mei, C. Ye, X. Zou, Z. Zhang, X. Liu, L. Wei, S. A. Truelove, T. Zhang, W. Gao, C. Cheng, X. Tang, X. Wu, Y. Wu, B. Sun, S. Huang, Y. Sun, J. Zhang, T. Ma, J. Lessler, and T. Feng. Epidemiology and transmission of COVID-19 in 391 cases and 1286 of their close contacts in Shenzhen, China: a retrospective cohort study. *Lancet Infectious Diseases*, 2020.

[17] A. Bittau, U. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnes, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[18] J. Blocki, A. Blum, A. Datta, and O. Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2013.

[19] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. M. Kiddon, J. Konecny, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. Towards federated learning at scale: System design. In *Proceedings of Machine Learning and Systems*, 2019.

[20] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. https://eprint.iacr.org/2011/277.

[21] R. Canetti, Y. T. Kalai, A. Lysyanskaya, R. L. Rivest, A. Shamir, E. Shen, A. Trachtenberg, M. Varia, and D. J. Weitzner. Privacy-preserving automated exposure notification. *IACR Cryptol. ePrint Arch*, 2020.

[22] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 1981.

[23] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[24] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2015.

[25] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[26] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2015.

[27] G. Danezis. Statistical disclosure attacks. In *IFIP International Information Security Conference*, pages 421–426. Springer, 2003.

[28] 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*, 2013.

[29] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. P. B. authors contributed equally). Gluon: A communication optimizing framework for distributed heterogeneous graph analytics. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[30] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *NIPS*, 2017.

[31] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, 2004.

[32] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[33] C. Dwork, A. Karr, K. Nissim, and L. Vilhuber. On privacy in the age of covid-19. *Journal of Privacy and Confidentiality*, 2020.

[34] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2006.

[35] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Theory of Cryptography Conference (TCC)*. 2006.

[36] C. Dwork and A. Roth. *The Algorithmic Foundations of Differential Privacy*. Now Publishers Inc, Aug. 2014.

[37] A. Endo, S. Abbott, A. J. Kucharski, S. Funk, et al. Estimating the overdispersion in covid-19 transmission using outbreak sizes outside china. *Wellcome Open Research*, 2020.

[38] U. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preservingordinal response. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[39] G. Fanti, V. Pihur, and U. Erlingsson. Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries. *arXiv:1503.01214 [cs]*, 2015.

[40] K. B. Frikken and P. Golle. Private social network analysis: How to assemble pieces of a graph privately. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, 2006.

[41] D. Froelicher, P. Egger, J. Sousa, J. L. Raisaro, Zhicong Huang, C. Mouchet, B. Ford, and J.-P. Hubaux. UnLynx: A Decentralized System for Privacy-Conscious Data Sharing. *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2017.

[42] D. Froelicher, P. Egger, J. S. Sousa, J. L. Raisaro, Zhicong Huang, C. Mouchet, B. Ford, and J.-P. Hubaux. UnLynx: A Decentralized System for Privacy-Conscious Data Sharing. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2017.

[43] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *arXiv preprint arXiv:1904.07162*, 2019.

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

[45] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[46] K. Gopinath and V. H. Gupta. An extended verifiable secret redistribution protocol for archival systems. In *Proceedings. The First International Conference on Availability, Reliability and Security*, 2006.

[47] J. Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

[48] 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*, 2020.

[49] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proceedings of the USENIX Security Symposium*, 2011.

[50] M. Hay, G. Miklau, D. Jensen, D. Towsley, and P. Weis. Resisting structural re-identification in anonymized social networks. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[51] I. Hoque and I. Gupta. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013.

[52] 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*, 2020.

[53] KU Leuven COSIC. SCALE-MAMBA. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[54] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2017.

[55] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[56] A. Kwon, D. Lu, and S. Devadas. {XRD}: Scalable messaging system with cryptographic privacy. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[57] M. S. Lam, S. Guo, and J. Seo. Socialite: Datalog extensions for efficient social network analysis. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, 2013.

[58] R. Laxminarayan, B. Wahl, S. R. Dudala, K. Gopal, S. Neelima, K. J. Reddy, J. Radhakrishnan, J. A. Lewnard, et al. Epidemiology and transmission dynamics of covid-19 in two indian states. *Science*, 2020.

[59] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[60] K. Liu and E. Terzi. Towards identity anonymization on graphs. In *Proceedings of the ACM SIGMOD Conference*, 2008.

[61] Y. Liu, R. M. Eggo, and A. J. Kucharski. Secondary attack rate and superspreading events for sars-cov-2. *The Lancet*, 2020.

[62] J. O. Lloyd-Smith, S. J. Schreiber, P. E. Kopp, and W. M. Getz. Superspreading and the effect of individual variation on disease emergence. *Nature*, 2005.

[63] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *Uncertainty in Artificial Intelligence*, 2010.

[64] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD Conference*, 2010.

[65] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C.Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD Conference*, 2010.

[66] S. Mazloom and S. D. Gordon. Secure computation with differentially private access patterns. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[67] P. Mittal, M. Wright, and N. Borisov. Pisces: Anonymous communication using social networks. *arXiv preprint arXiv:1208.6326*, 2012.

[68] 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 Med*, 2008.

[69] A. Narayan and A. Haeberlen. DJoin: Differentially private join queries over distributed databases. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[70] J. P. Near, D. Darais, C. Abuah, T. Stevens, P. Gaddamadugu, L. Wang, N. Somani, M. Zhang, N. Sharma, A. Shan, and D. Song. Duet: An expressive higher-order language and lineartype system for statically enforcing differential privacy. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2019.

[71] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[72] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[73] 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*, 2019.

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

[75] 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*, 2020.

[76] V. Pihur, A. Korolova, F. Liu, S. Sankuratripati, M. Yung, D. Huang, and R. Zeng. Differentially-private "draw and discard" machine learning. *ArXiv*, 2018.

[77] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The loopix anonymity system. In *Proceedings of the USENIX Security Symposium*, 2017.

[78] R. Pung, C. J. Chiew, B. E. Young, S. Chin, M. I. Chen, H. E. Clapham, A. R. Cook, S. Maurer-Stroh, M. P. Toh, C. Poh, et al. Investigation of three clusters of covid-19 in singapore: implications for surveillance and response measures. *The Lancet*, 2020.

[79] J.-F. Raymond. Traffic analaysis: Protocols, attacks, design issues, and open problems. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[80] E. Roth, D. Noble, B. H. Falk, and A. Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[81] E. Roth, H. Zhang, A. Haeberlen, and B. C. Pierce. Orchard: Differentially private analytics at scale. In *Proceedings of the USENIX*

*Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[82] A. Shamir. How to share a secret. *Commun. ACM*, 1979.

[83] C. Troncoso, M. Payer, J.-P. Hubaux, M. Salathé, J. Larus, E. Bugnion, W. Lueks, T. Stadler, A. Pyrgelis, D. Antonioli, et al. Decentralized privacy-preserving proximity tracing. *arXiv preprint arXiv:2005.12273*, 2020.

[84] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[85] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[86] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2015.

[87] A. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[88] Y. Zhang, C. Wang, D. Pujol, J. Bater, M. Lentz, A. Machanavajjhala, K. Nayak, L. Vasudevan, and J. Yang. Poirot: private contact summary aggregation. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020.

[89] E. Zheleva and L. Getoor. Preserving the privacy of sensitive relationships in graph data. In *Proceedings of the 1st ACM SIGKDD International Conference on Privacy, Security, and Trust in KDD*, 2008.

[90] B. Zhou and J. Pei. Preserving privacy in social networks against neighborhood attacks. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, 2008.

[91] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.