Honeycrisp: Large-Scale Differentially Private Aggregation Without a Trusted Core

Edo Roth
University of Pennsylvania

Daniel Noble
University of Pennsylvania

Brett Hemenway Falk
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Abstract

Recently, a number of systems have been deployed that gather sensitive statistics from user devices while giving differential privacy guarantees. One prominent example is the component in Apple's macOS and iOS devices that collects information about emoji usage and new words. However, these systems have been criticized for making unrealistic assumptions, e.g., by creating a very high "privacy budget" for answering queries, and by replenishing this budget every day, which results in a high worst-case privacy loss. However, it is not obvious whether such assumptions can be avoided if one requires a strong threat model and wishes to collect data periodically, instead of just once.

In this paper, we show that, essentially, it is possible to have one's cake and eat it too. We describe a system called Honeycrisp whose privacy cost depends on how often the data *changes*, and not on how often a query is asked. Thus, if the data is relatively stable (as is likely the case, e.g., with emoji and word usage), Honeycrisp can answer periodic queries for many years, as long as the underlying data does not change too often. Honeycrisp accomplishes this by using a) the sparse-vector technique, and b) a combination of cryptographic techniques to enable global differential privacy *without* a trusted party. Using a prototype implementation, we show that Honeycrisp is efficient and can scale to large deployments.

1 Introduction

Differential privacy [26] has become the gold standard for performing analysis on sensitive data while giving strong, provable privacy guarantees. A common way to answer queries about a data set in a differentially private way is to 1) compute the exact answer to the query, and to then 2) add a carefully calibrated amount of noise to the answer before returning it to the client. There is now a substantial literature on differential privacy, both on the theoretical foundations [28] and on practical implementations [20, 55, 62, 68]. There are also several large-scale deployments, including one in Google's Chrome web browser [32] and another in Apple's iOS devices [8].

For concreteness, let us consider one specific use case from Apple's deployment [5] in a bit more detail. To get a better sense of how popular each emoji is, Apple devices record an event every time the user types an emoji – assuming the user has opted in – and temporarily store the events, with appropriate noise added in, locally on the device. Then, once in a while, the device samples a subset of these events and sends them to Apple's servers [8], where they are aggregated with events from other devices and then analyzed.

Existing deployments like the one described above face two important challenges. The first has to do with the way the data is collected. Much of the early literature assumes a trusted data curator, who collects the data in the clear, aggregates it, and, as the final step, adds noise to the (precise) answer of the query. We refer to this model as *global differential privacy (GDP)*. In Apple's deployment, however, noise is added locally by each user *before* the contributions are collected by Apple. This is called *local differential privacy (LDP)* [32]. Adding noise locally, before aggregation, is necessary for user privacy, since otherwise Apple would have access to the user data in the clear and could be compelled to collect and reveal the data of individual users. (Apple receives thousands of requests for data from law enforcement every year [6].)

Although LDP is clearly better for privacy, it also adds considerably more noise to the overall data set and thus reduces the accuracy that can be achieved from comparable queries. The differential privacy literature reasons about this tradeoff between privacy and utility by assuming a *privacy budget* that reflects the users' privacy expectations; it then assigns a "cost" to each query that reflects the amount of information the query can leak and that must be deducted from the budget each time the query is asked. In general, LDP requires a much larger privacy budget than GDP because, to achieve similarly accurate results, the amount of perturbation of each data point must be significantly lower.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada
© 2019 Copyright held by the authors. Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6873-5/19/10. . . $15.00
https://doi.org/10.1145/3341301.3359660

The second challenge has to do with the fact that new data is uploaded regularly (e.g., daily). Regular updates are necessary because user behavior can change over time and Apple or Google would presumably like to track such changes; however, it also means that, even if the answers are appropriately noised, the noise terms from repeated queries will eventually cancel out as more and more queries are answered, revealing statistics about the user's behavior. This leakage further exacerbates the first problem: to get the same level of accuracy, the privacy budget would need to be even larger! If one stops answering queries once the budget is exhausted, this approach provides strong guarantees. However, no finite budget would be enough to support periodic queries indefinitely, which is why Apple opted to replenish the budget *every day* [72]. This would be reasonable if 1) users were comfortable with potentially revealing emojis they typed yesterday, or 2) the emoji usage by the same user on two different days were completely uncorrelated; however, neither seems like a realistic assumption.

In this paper, we propose a possible way out of this dilemma. We present a system called Honeycrisp that can sustainably run queries like the one from Apple's deployment while protecting user privacy *in the long run*, as long as the underlying data does not change too often – which is likely, e.g., in the case of emoji usage patterns. Honeycrisp accomplishes this with a combination of two key insights. The first is a new threat model, which we call *occasionally Byzantine + mostly correct (OB+MC)*, and which we have specially tailored to large-scale deployments with millions of users, such as Apple's or Google's. In contrast to prior work, such as Prochlo [13], UnLynx [36], or Outis [21], we do not assume powerful third parties that could take on a substantial amount of work: with millions of users, any substantial involvement would require a lot of resources – perhaps even a data center, which few parties can afford. On the other hand, we assume that the adversary can compromise at most a small fraction (say, 1–5%) of the users' devices. This is substantially lower than the usual 1/2 or 1/3, but, at the scale we are targeting, it would still mean far more corrupted devices than are found, e.g., in a typical botnet.

Our second insight is that, in this model, we can use a cocktail of cryptographic techniques – specifically, multiparty computation (MPC) [75] and a form of homomorphic encryption – to efficiently implement global differential privacy *at scale*, which enables us to leverage the sparse vector technique (SVT) [27, 66] from the differential privacy literature. We introduce a technique we call *collect-and-test (CaT)* that can accomplish this, and we present a concrete set of algorithms that implement CaT, along with a security proof. Interestingly, our approach *does not require a trusted party at all*. Even the system operator itself (e.g., Apple or Google) does not need to be trusted; Honeycrisp uses it only to facilitate the computation by providing resources, such as computation and bandwidth.

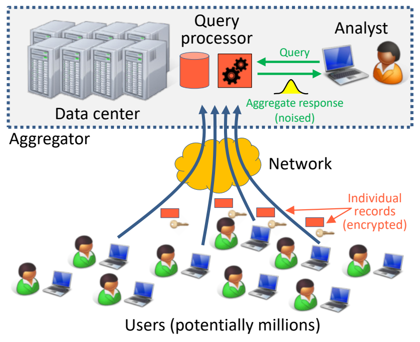

Query processor
Analyst
Query
Aggregate response (noised)
Data center
Aggregator
Network
Individual records (encrypted)
Users (potentially millions)

Figure 1. Scenario.

Using a prototype implementation, we demonstrate that Honeycrisp can support a form of aggregation that is common in both Apple's and Google's current deployments and would be fast enough to run at scale, with billions of user devices. With a billion devices and our choices for the cryptographic building blocks, the aggregator would need to provide roughly 1.2 MB of bandwidth per user per query, and less than 50 cores; most user devices would need to provide about 1.2 MB of traffic and about 60 seconds of computation time, although a tiny, randomly chosen set of devices would need to provide substantially more. We also show that, with comparable security and privacy, a LDP system would exhaust a typical privacy budget after only 91 days, whereas Honeycrisp could run for up to ten years. In summary, our contributions are:

- the collect-and-test technique (Section 2);
- the design of Honeycrisp (Section 3);
- a prototype implementation (Section 4); and
- an experimental evaluation (Section 5).

2 Overview

Figure 1 illustrates the scenario we are considering in this paper. There is a very large number of users (e.g., all iPhone and MacBook owners, or all Chrome users), as well as one central *aggregator* $\mathcal{A}$ – e.g., Apple or Google. Each user regularly collects some sensitive information on her device that she wishes to make available to the aggregator for analysis, provided that her privacy can be guaranteed. The aggregator has substantial computational resources – e.g., a data center – and is able to collect the uploaded data from the devices, as well as perform some cryptographic operations. The aggregator also has at least one *analyst*, who would like to issue queries about the collected data; for instance, one possible query could be a count-mean sketch of emojis or new words that are not yet in a dictionary, as in [8].

2.1 The OB+MC threat model

To provide strong protections, we would like to be robust not just to honest-but-curious (HbC) behavior, but rather to actual Byzantine faults. At smaller scales, the standard threat model for this setting would be to assume that a certain fraction (usually one third) of all nodes can be Byzantine. However, this seems overly pessimistic for our scenario, for two reasons.

Aggregator: Occasionally Byzantine (OB). First, the enormous size and prominent position of the aggregator would subject it to a lot of scrutiny (from the press, the users, etc.), so it is not likely to be Byzantine for long. It can very well be Byzantine for brief periods, however – for instance, due to misbehavior by rogue employees. Because of this, even a well-intentioned aggregator might not "trust itself" to always behave correctly, and might wish to design the system to limit the damage it could do during any Byzantine periods.

Users: Mostly Correct (MC). Second, if the number of users is very large (e.g., the 1.3 billion macOS/iOS devices [4]), it seems unlikely that an adversary could compromise a large fraction of them. This is different from, say, BFT: in a replica set of 4–7 nodes, compromising 1/3 of the system means just one or two nodes. But at the scale of the Apple ecosystem, even compromising 3% would mean about 39 *million* nodes, which is much larger than, e.g., a typical botnet.

We refer to these assumptions as the *OB+MC* threat model, to distinguish it from the classic Byzantine fault model and its 1/3 failure threshold. To reiterate, we assume that a) the aggregator is HbC when the system starts and usually remains HbC, except for occasional periods of Byzantine behavior, and that b) a *small* fraction of the devices, on the order of a few percent, is Byzantine as well. Notice that the latter requires that the aggregator refrain from building back doors into its devices, so that, during its Byzantine periods, it cannot – or will not [7] – change the devices' software.

We explicitly do *not* assume the existence of a trusted third party that is willing to be actively involved. If a party is available that can be trusted with some very limited tasks, such as generating random bits, Honeycrisp can use it for efficiency (as described in [67, §B]), but it is not required.

Goals: Our primary goal is to protect user privacy. When the aggregator is behaving correctly, we also ensure integrity (that is, accurate query results), but we drop this second goal during the aggregator's Byzantine periods. This seems reasonable, since the aggregator is the beneficiary of the collected data and can only harm itself by misbehaving.

2.2 Background: Differential privacy

We begin by providing some brief background. Differential privacy is a property of randomized functions that take a database as input, and returns an aggregate output. Informally, a function is differentially private if changing any single row in the input database results in *almost* no change in the output. If we view each row as consisting of the data of a single individual, this means that any single individual has a statistically negligible effect on the output. This guarantee is quantified in the form of a parameter, ϵ, which corresponds to the amount that the output can vary based on changes to a single row. Formally, for any two databases d_1 and d_2 that differ only in a single row, we say that f is *ϵ-differentially private* if, for any set of outputs R,

$$Pr[f(d_1) \in R] \leq e^{\epsilon} \cdot Pr[f(d_2) \in R]$$

In other words, a change in a single row results in at most a multiplicative change of e^{ϵ} in the probability of any output, or set of outputs.

The standard method for achieving differential privacy for numeric queries is the *Laplace mechanism* [26], which involves two steps: first calculating the *sensitivity*, s, of the query – which is how much the un-noised output can change based on a change to a single row – and second, adding noise drawn from a Laplace distribution with scale parameter s/ϵ; this results in ϵ-differential privacy. Differential privacy is also compositional, that is, if we evaluate two functions f_1 and f_2 that are ϵ_1- and ϵ_2-differentially private, respectively, publishing the results from both functions is at most $(\epsilon_1 + \epsilon_2)$-differentially private. This property is often used to keep track of the amount of private information that has already been released: we can define a *privacy budget ϵ_{max}* that corresponds to the maximum loss of privacy that the subjects are willing to accept, and then deduct the "cost" of each subsequent query from this budget until it is exhausted. For a detailed discussion of ϵ_{max}, see, e.g., [43].

2.3 Background: The Sparse-Vector Technique

The Laplace mechanism, in combination with a finite privacy budget, cannot support repeated queries indefinitely, since the budget will eventually be exhausted. However, the following, different mechanism allows an analyst to make regular, repeated queries without significantly reducing the privacy budget with each query. The analyst does not ask for $f(x)$ directly; instead, she provides a "guess" $\hat{f}$ and asks only whether $|f(x) - \hat{f}| > \hat{T}$, where $\hat{T}$ is some small, noised threshold. The actual value $f(x)$ is then released *only* if the answer is yes. This is called the *sparse-vector technique (SVT)* [27, 28, 66].

The SVT has the key advantage that the privacy budget needs to be charged significantly only if the answer to the threshold query is yes – that is, if the answer to the query does differ from the analyst's guess. (Intuitively, the reason is that the analyst does not really learn anything new if the guess was correct.) A small charge is necessary even if the answer is no, but, via advanced composition [29], this charge can be "prepaid" at the beginning and amortized over a large number of queries. Thus, the privacy budget is depleted mostly in proportion to how frequently the data *changes*, with an additional logarithmic decay to account for negative answers and the possibility of error in threshold comparison. The details for this privacy budget improvement are discussed in detail in Section 5.2. In our motivating scenario, such changes (different emoji preferences, or appearance of new, previously unknown words) are likely to be rare. Thus, the SVT enables the analyst to run the system for much longer, or even indefinitely, without assuming that the users are willing to tolerate high worst-case information leaks.

2.4 Strawman solutions

Collect the data unencrypted: One way to implement the SVT would be to simply have the aggregator collect all the data unencrypted, and to perform the thresholding at the aggregator. In our threat model, this is not an option: the aggregator could become Byzantine at any time and would then be able to leak the plain-text information of any user.

Use large-scale MPC: Another way would be to implement the SVT using a multi-party computation (MPC) between all the devices. Each device could input its local data, and the MPC could then aggregate the data, do the thresholding, and then either release the new answer or indicate that the answer has not changed. However, generic MPC is known to scale very poorly with the number of participants: efficient MPC techniques are available for two parties (e.g., [48]) and some can handle dozens of parties (e.g., [73]) but we are not aware of any technique that could be used for a billion parties.

Use small-scale MPC: The MPC could also be performed at a smaller scale, e.g., between the aggregator and one device, or a small subgroup of devices. However, this is risky because we have assumed that the aggregator is capable of small-scale collusion and/or a small-scale Sybil attack – for instance, they could manufacture a few extra devices, keep them, and always perform the MPC with these devices. Also, it is not clear how the data would be aggregated: individual devices (e.g., phones and tablets) are not likely to be capable of receiving and processing millions of records from other users, nor can they necessarily be trusted with this information.

2.5 Our approach: Collect-and-Test

We now sketch our actual approach, which we call *collect-and-test (CaT)*. CaT proceeds in the following three phases.

Setup phase: In the first phase, CaT uses a sortition scheme (Section 3.2) to randomly and accountably choose a *committee*, which is a small subset of devices. The committee then uses MPC to generate a keypair for an additively homomorphic cryptosystem. The private key is secret-shared, and the shares are kept on the committee's devices, whereas the public key is endorsed by the devices and sent to the aggregator (Section 3.3).

Collect phase: In the next phase, the aggregator uses its resources to distribute the public key and the endorsements to all the devices; each device verifies the endorsements (Section 3.4), then encrypts her data with this key, and sends the ciphertext back to the aggregator, along with a zero-knowledge proof that the encrypted plaintext is formatted correctly and in the right range. (Note that the aggregator does *not* know the private key for the cryptosystem and thus cannot perform these checks on the plaintext directly!) Finally, the aggregator verifies the range proofs, aggregates the ciphertexts using the homomorphic property of the cryptosystem, and thus obtains a single ciphertext that contains the (precise, un-noised) sum of the individual records (Section 3.5).

Test phase: Finally, the aggregator sends the (single) aggregate ciphertext back to the committee, along with its "guess" for the plaintext value. The committee members input their key shares, the guess and the ciphertext into another MPC, which combines the shares, recovers the private key (Section 3.6), and decrypts the ciphertext to obtain the precise sum. The MPC then generates random bits to noise the sum (Section 3.7) and compares the result to the aggregator's "guess" (Section 3.8). If the difference is larger than the threshold, the MPC outputs the true result; otherwise it outputs a default value to indicate that the result is close to the guess.

2.6 Challenges

At first glance, it may seem that the key ideas are only in the approach (e.g., the applicability of the SVT and homomorphic encryption), and that an implementation of CaT could simply consist of a few standard cryptographic building blocks. However, there are also two subtle technical challenges. First, although the aggregator cannot directly read the encrypted data, it can attempt to infer the data in other ways – e.g., by leaving out some ciphertexts while computing the aggregation, and/or by fabricating Sybil identities that will adaptively choose the ciphertexts they contribute (for instance, identical to the ciphertext of a specific user whose data the aggregator wants to learn). To address this, we have developed a verifiable aggregation protocol for the Collect phase that can ensure that the aggregator 1) includes the ciphertext of each user exactly once, 2) computes the aggregation correctly, and 3) can include at most a small fraction of malicious (but non-adaptive) inputs. The second challenge is scalability: with easily a billion participants that each have only very limited resources, we must design the protocol very carefully to avoid overwhelming individual participants.

3 The Honeycrisp system

Next, we describe a concrete system called Honeycrisp that implements the CaT approach in the OB+MC model. Honeycrisp relies on the following assumptions:

1. Each device i has a locally generated keypair σ_i/π_i for signing messages; the aggregator can check whether each public key π_i belongs to a valid device.
2. There is a once-off randomness beacon – an independent party P that can be trusted to generate a single random string, B_0, when the system is first launched.
3. All devices know an upper bound N_{max} and a lower bound N_{min} of the number of potential participating devices in the system.
4. There is an immutable bulletin board B that the aggregator can use to broadcast a small amount of data to all devices.

5. Devices can use an external, time-stamped channel X to report the aggregator if it behaves maliciously.
6. Secure, authenticated, point-to-point channels can be established from each device to a) the aggregator, and b) a small number of other devices.
7. There is an upper bound f ($\approx$ 1–5%) on the fraction of participating devices that may be malicious, collude with each other, or collude with the aggregator.
8. There is an upper bound g on the probability that an honest device goes offline while participating in a round.
9. There exists an efficient hash function that is indistinguishable from a random oracle.

For instance, in the case of Apple, these assumptions could be satisfied by 1) the Secure Enclave coprocessor in recent devices; 2) an existing random number service, such as `random.org`, or a widely respected party, such as the EFF; and 3) public estimates on the number of devices sold [59], and/or self-reported statistics on installed base of iPhone users [74]; again, only an imprecise range is necessary. 4) could be any of several (free, centralized) "bulletin boards", such as Wikipedia, StackExchange, or Reddit; only the aggregator needs to post transactions, and only a small number of times per round, so neither cost nor latency should be an issue. For 5), if users have evidence that the aggregator has acted maliciously, they could post this evidence in an online forum (Twitter, Wikipedia, ...) or give it to the press. 6) could be satisfied with TLS channels, in combination with NAT traversal techniques [34]; 7) seems plausible given experience with existing deployments (see 2.1); 8) seems plausible given the always-on nature of modern devices (which is being leveraged, e.g., for push notifications), and 9) is a common model for cryptographic protocols. For additional details about our assumptions and ways to satisfy them, please see [67, §C.1].

We also make a simplifying assumption, which is that most users have only one device, and that it is therefore sufficient to provide a per-device privacy guarantee. However, Honeycrisp can easily be changed to give a per-*user* privacy guarantee instead – by selecting a single device for each user (e.g., based on AppleID) and by having only this device respond to queries, using data from that user's entire set of devices.

3.1 Committees and rounds

Recall from Section 2.5 that there is a committee of C devices that holds the shares of the private key for the homomorphic encryption, and that also maintains the privacy budget. Since the committee is composed of regular devices, it would be very burdensome to require the same devices always perform the role of the committee. Hence, Honeycrisp segments its execution into discrete *rounds*, and it randomly appoints a new committee for each round.

The security of the scheme is contingent on the dependability of this committee. Since we cannot trust individual devices, any action that could cause sensitive data to leak ("privacy failure"), such as making decisions on behalf of the committee or reconstructing the secret key, must require a large subset of, say, A members. But A cannot be too large either, otherwise it can happen that some queries do not receive an answer ("liveness failure") because some committee members – say, B devices – go offline during a round.

In our design, we chose $A = \frac{2}{5}C$ and $B = \frac{1}{5}C$. Using a probabilistic argument, we can show that, if up to $f = 3\%$ of the devices are malicious and the system runs one round per day for ten years, $C \geq 29$ is sufficient to prevent privacy failures with probability 99.999%, while ensuring that at least 95% of the queries are answered successfully (with an unsuccessful query simply resulting in a re-run in the subsequent round). We provide more details in Section 5.4 and the full analysis in [67, §C.3].

3.2 Setup phase: Sortition

Next, we show how the committee can be selected in such a way that an adversary cannot influence or predict the selection. This particular building block has appeared in several earlier systems, including Algorand [37] and RandHound/RandHerd [71]; here, we adapt the approach from Algorand because, unlike RandHound/RandHerd, it can scale to millions of participants.

Briefly, the protocol works as follows. Each round i has a "block" B_i of random bits. The blocks are usually uniformly random from $\mathcal{A}$'s perspective, and $\mathcal{A}$ can only manipulate them within strict limits. B_i determines the committee, as well as a "leader" L_i, as follows. First, each device signs three messages $(B_i, i, 0)$, $(B_i, i, 1)$, and $(B_i, i, 2)$. (The third element in these triples is just to ensure that the hashes of the messages are independent.) The committee then consists of the devices whose signatures on $(B_i, i, 0)$ have the lowest hash, the "leader" is the device whose signature of $(B_i, i, 1)$ has the lowest hash, and the next random number B_{i+1} equals the hash of the leader's signature of $(B_i, i, 2)$.

A detailed description of the protocol, which we call GET_NEW_COMMITTEE, is in the figure below. As part of the protocol, $\mathcal{A}$ maintains a Merkle tree [57] of an array of registered devices. This allows it to publish a constant-sized tree root that is bound to the state of the array at a given point in time, and subsequently to provide logarithmic-sized proofs that devices are in the committed array [9]. We assume that B_0 is a random number that is provided by a trusted source *after* the set of initial devices, $R_{-\infty}$, is already committed to by placing the tree root on the bulletin board B.

Every time a device sends $\mathcal{A}$ a message, $\mathcal{A}$ must send a signed acknowledgment of having received the specific message. If $\mathcal{A}$ fails to do so, the device reports through the reporting channel, X, that it has not yet received a message that is due from $\mathcal{A}$, giving $\mathcal{A}$ an opportunity to respond publicly. If she does not, the device reports that $\mathcal{A}$ has deviated from the protocol. This prevents $\mathcal{A}$ from ignoring devices, in particular devices that should be leaders or committee members.

GET_NEW_COMMITTEE

1. Each new device who wishes to join registers its public key with $\mathcal{A}$. A device is only eligible to be a leader or committee member if it has been registered for at least κ rounds or was an initial device. $\mathcal{A}$ adds each new key to the set R_i. $\mathcal{A}$ creates a Merkle tree of R_i and posts the root to the bulletin board. This will allow $\mathcal{A}$ to generate proofs $\mu_{i',j}$, that a device j is eligible for election in round i', by showing that $j \in R_t$ for some $t \leq i' - \kappa$.
2. Each device $j \in R_t$ for some $t \leq i - \kappa$ computes $\eta_{i,j,0} = \text{sign}_{sk_j}(B_i, i, 0)$ and sends it to $\mathcal{A}$.
3. $\mathcal{A}$ computes $h_{i,j,0} = \text{Hash}(\eta_{i,j,0})$ for each j. The devices with the C lowest $h_{i,j,0}$ form the committee. $\mathcal{A}$ posts the committee, along with their $\eta_{i,j,0}$ and $\mu_{i,j}$, to the bulletin board.
4. Each device j that is in set R_t for $t \leq i - \kappa$ computes $\eta_{i,j,1} = \text{sign}_{sk_j}(B_i, i, 1)$ and sends it to $\mathcal{A}$.
5. $\mathcal{A}$ computes $h_{i,j,1} = \text{Hash}(\eta_{i,j,1})$ for each j. The device with the minimum $h_{i,j,1}$ is the leader L_i. $\mathcal{A}$ posts $(L_i, \eta_{i,\,L_i,\,1}, \mu_{i,\,L_i})$ to the bulletin board.
6. The leader L_i sends $\eta_{i,\,L_i,\,2} = \text{sign}_{sk_{L_i}}(B_i, i, 2)$ to $\mathcal{A}$, who posts it on the bulletin board. Then $B_{i+1} = \text{Hash}(\eta_{i,\,L_i,\,2})$.
7. If the leader does not respond in time, then $B_{i+1} = \text{Hash}(B_i, i)$.
8. Each device j checks that:
   - If j is not on the committee, then $h_{i,j,0} > h_{i,k,0}$ for each committee member, k.
   - $\mu_{i,k}$ is correct for each committee member k.
   - For each committee member, k, $\eta_{i,k,0}$ is a correct signature for k.
   - If $j \neq L_i$ then $h_{i,j,1} > h_{i,L_i,1}$.
   - $\mu_{i,\,L_i}$ is correct.
   - $\eta_{i,\,L_i,\,1}$ and $\eta_{i,\,L_i,\,2}$ are correct signatures for L_i.

If any of these fail, the device sends the evidence of the failure to the reporting channel, X, and aborts the protocol.

A full proof of how this ensures the unlikeliness of a malicious committee is provided in [67, §C.3].

3.3 Setup phase: Key generation

Once a new committee has formed, the committee members must generate a new keypair (PK, SK) for the homomorphic cryptosystem that will be used to encrypt and aggregate the users' private data records for this round. As explained in 3.1, the secret key will be stored using a secret sharing scheme. It will remain safe, as long as there are fewer than $\frac{2}{5}C$ malicious committee members. Additionally, the scheme must be able to detect if malicious committee members attempt to introduce an error into the secret during reconstruction, provided fewer than $\frac{2}{5}C$ of them collaborate to attempt this. Lastly the scheme should allow for up to $\frac{1}{5}C$ committee members to go offline. We achieve this with Shamir Sharing [69], based on the Reed-Solomon code [65], with parameter $t = \lfloor \frac{2}{5}C \rfloor$.

The following protocol, KEY_GEN, is performed within the MPC to securely generate a key-pair:

KEY_GEN

1. Choose $(PK, SK) \overset{r}{\leftarrow} KeyGen()$
2. Publicly reveal PK to all participants.
3. Distribute SK using a secret-sharing scheme that detects errors when there are fewer than $\frac{2}{5}C$ errors and is secure against up to $\frac{1}{5}C$ erasures.

3.4 Collect phase: Querying

Honeycrisp may run for a long time, and during that time the needs of the aggregator could change; thus, it could be problematic to hard-code a specific query, or set of queries, in the design. Instead, Honeycrisp can optionally support a simple query language that can be used to specify arbitrary queries over the data that is available at each device. For instance, the aggregator could ask for a count-mean sketch of emoticons today, and a count of the devices that have shut down because of low battery [5] tomorrow. The question of what to include in the "database" that is available for querying is up to the operator; users could also be allowed to enable or disable certain items based on their own preferences.

Since Honeycrisp relies on an additively homomorphic cryptosystem for aggregating the collected records, not all queries can be supported. However, we *can* support counts and sums, as long as we maintain queries that are 1-sensitive for differential privacy purposes. For instance, Honeycrisp can easily compute the number of devices that have a given property, make histograms or count-mean/count-min sketches, and can sum or average values from a group of devices. These types of queries boil down to two steps: the first, which we call the *map step*, maps each record in the data set to a vector of numeric values (or even a single value), and the second, which we call the *sum step*, then adds up all the vectors to produce the final output.

To verify that a proposed query has a finite "privacy cost" that is within the remaining privacy budget, the committee must be able to determine the sensitivity of the query – that is, the amount by which the answer can change if a single person's data is added or removed. We can enable this by writing the queries in a language such as Fuzz [39], which comes with a static analysis that bounds the sensitivity.

Once the committee has verified that the remaining privacy budget is sufficient for the proposed query, the honest committee members sign a query authorization certificate that includes the public key generated in Section 3.3, the query specification, the remaining privacy budget, and the current round, and they upload it to the aggregator, which distributes it to the other devices. The other devices verify that the certificate has been signed by at least $\frac{2}{5}$ of the current committee (whose membership they know from Section 3.2); if so, they accept the included public key and query.

3.5 Collect phase: Aggregation

Once a device has received the certificate and the query from the aggregator, and once it has verified the certificate, it locally performs the query's map step – using the data on that particular device – and obtains a vector of numeric values.

At first, it may seem that the device can simply encrypt the values using the homomorphic cryptosystem and send them to the aggregator. However, this is not enough to guarantee privacy. While $\mathcal{A}$ will not be able to learn any information from the ciphertext itself, $\mathcal{A}$ may, during a Byzantine period, send to the committee an incorrect aggregation, such that the query result exposes sensitive user information. For instance, if $\mathcal{A}$ used the additively homomorphic property to multiply a single device's input x_i by a sufficiently large constant, then the result of the query would allow conclusions about x_i, since the Laplace noise will be "too small" to hide such a large contribution. Alternatively, $\mathcal{A}$ could create Sybil identities and choose the inputs of these identities to be x_i as well – which $\mathcal{A}$ can do because the ciphertext c_i is uploaded to it. To prevent attacks such as these, we would like $\mathcal{A}$ to prove that 1) each honest device's input affects only its own ciphertext, and that 2) the summation was correctly computed.

To prevent adaptive choices of ciphertexts, Honeycrisp requires that all inputs to the summation be committed to before any are revealed. It also requires that the summation process is checked. Since the number of devices is too large for the entire summation to be checked by any device, $\mathcal{A}$ generates, and commits to, an object we call a *summation tree* that contains the inputs and partial sums. This is done using the AGGREGATE protocol below.

AGGREGATE

1. Each device holds a key-pair (σ_i, π_i) for a signature scheme and a private input x_i.
2. Each device computes $c_i = Enc_{PK}(x_i)$
3. Each device generates a commitment to (c_i, π_i), namely $t_i = \text{Hash}(r_i \,||\, c_i \,||\, \pi_i)$, where $r_i \overset{r}{\leftarrow} \{0,1\}^{128}$. The device sends (π_i, t_i) to $\mathcal{A}$.
4. $\mathcal{A}$ sorts pairs (π_i, t_i) by π_i to form an array of tuples *Commit*. $\mathcal{A}$ generates a Merkle tree M_C from array *Commit* and publishes the root to B.
5. Each device generates a zero-knowledge proof, z_i, that the plaintext x_i that corresponds to the ciphertext c_i is in the required range.
6. Each device sends (π_i, c_i, r_i, z_i) to $\mathcal{A}$.
7. $\mathcal{A}$ checks the message. If either the proof, z_i, or the commitment, $t_i = \text{Hash}(r_i||c_i||\pi_i)$, is wrong, they ignore the message.
8. $\mathcal{A}$ generates a summation tree, S. The leaves are set to be $S(0, i) = (\pi_i, c_i, r_i)$ if $\mathcal{A}$ received a correct message from a device and $(\pi_i, \perp)$ otherwise. Each non-leaf vertex has two children and a ciphertext that is the sum of its children's ciphertexts.
9. $\mathcal{A}$ serializes all vertices of S into an array and then publishes a Merkle tree M_S of this array, as well as the root of the summation tree S, (the sum of all ciphertexts). To each device sent a correct leaf, $\mathcal{A}$ also sends a proof that this leaf is in M_S.

Each device checks a small random portion of this tree, using the CHECK_AGGREGATION protocol. Devices can check that an item is in the set by asking $\mathcal{A}$ to sending a membership proof for the item which consists of $\lceil \log N \rceil + 1$ hashes from the Merkle tree [57]. If no device reports a problem, this means that, with high probability ($\geq 99\%$, based on a security parameter s), the entire summation is correct.

Notice that the protocol also requires the devices to prove, in zero knowledge, that their inputs are in the correct range – e.g., using a zk-SNARK [10]. This step is not necessary for privacy, but it is necessary for integrity: without it, a single malicious device could render the entire query result useless by encrypting and submitting a very large random value.

3.6 Test phase: Key recovery

At the end of the collect phase, the aggregator has obtained the encrypted true result of the query. Next, the result must be decrypted and compared to the analyst's "guess." Since no individual party can be trusted with the full private key, the committee must run another MPC to do the decryption.

The aggregator submits the vector of ciphertexts and the analyst's guess(es) to the committee. The committee members then launch a multi-party computation to which they each input 1) their share of the private key, 2) the ciphertext from the aggregator, 3) the analyst's guess, and 4) a threshold difference, below which variation of the guess from the actual result will not be revealed. Inside this computation, the private key is reconstructed from the shares, and is then used to decrypt the ciphertext(s). If too many committee members have gone offline since the beginning of the round or now refuse to participate, the MPC run fails and the aggregator does not receive a response to her query for that round.

3.7 Test phase: Noising

Once the encrypted sums have been decrypted inside the MPC, some noise must be added to the plaintext values *before* they are compared to the analyst's guess. (This is part of

CHECK_AGGREGATION

Each device:

1. Verifies that $N \leq N_{max}$, and that the value $Commit_i$ it sent to $\mathcal{A}$ appears in M_C.
2. Chooses a random $v_{init} \in [0, N-1]$. Then for $i \in [v_{init}, v_{init}+s]$ mod N, verifies that:
   - $Commit_i$ appears in M_C
   - $S(0,i) = (\pi_i, \perp)$ or (π_i, c_i, r_i).
   - If $c_i, r_i \neq \perp$, checks $t_i = H(r_i || c_i || \pi_i)$.
   - $S(0,i)$ appears in M_S.

   Then for $i \in [v_{init}, v_{init}+s)$ mod N checks that $\pi_i < \pi_{i+1}$ (except if $i = N-1$).
3. Chooses s distinct non-leaf vertices of S. To reduce redundancy, this should include the (roughly $s/2$) vertices whose children the device has already obtained from the previous step. The remaining vertices should be chosen randomly from vertices that do not have leaves as children. For each vertex, the device verifies:
   - That the vertex's ciphertext is indeed the sum of its childrens' ciphertexts.
   - That the vertex and its children are in M_S.

If any check fails, the device publicly publishes to X the proof that $\mathcal{A}$ behaved maliciously (signed claims from $\mathcal{A}$ that are inconsistent).

the SVT.) The noise must be drawn from a distribution which gives correct differential privacy guarantees, and there must not be a way for a malicious committee member to bias the noise in any way. Often, random noise drawn from a Laplace distribution with parameter $(\Delta f/\epsilon)$ is used to support Δf-sensitive queries, as this guarantees $(\epsilon, 0)$ differential privacy. In our case, we simply support 1-sensitive queries to make use of the sparse vector mechanism, so we fix $\Delta f = 1$. The amount of noise will be a fixed amount set by the MPC in Honeycrisp based on the pre-determined privacy budget, such that no party (either the aggregator or the committee members) has any ability to affect the privacy guarantees. One additional concern is the existence of floating-point vulnerabilities that may arise from irregularities in existing implementations of the Laplacian mechanism that create porous (and thus attackable) distributions. Thus, the noise generation must be carefully implemented (for instance, with a snapping mechanism as described in [58]) to ensure differential privacy and to prevent such attacks.

3.8 Test phase: Thresholding

Finally, the noised results are compared to the analyst's guess. Once again, this must be done within the multi-party computation, to prevent individual devices from "leaking" the result to the aggregator. Somewhat counter-intuitively, such a leak would be problematic even after noise has been added: the privacy budget is not substantially charged if the analyst's guess was correct, so, if the data is stationary, the analyst could run very many queries "for free", average out the noise, and then use the precise result to infer the individual inputs. This requirement means that Honeycrisp cannot use a generic threshold cryptosystem [25] but instead must use more powerful MPC-based approach.

If the difference between the guess and the noised result is larger than the threshold, the computation outputs the noised result, and otherwise a default value to indicate that the guess was approximately correct. In the former case, the committee members deduct the "cost" of the query from the privacy budget and report the noised result back to the aggregator; in the (common) latter case, they simply report the outcome and decrement the large number of "prepaid" negative answers (see Section 2.3) but leave the budget itself unchanged.

3.9 Security analysis

A full formal definition of the security requirements, as well as proof that Honeycrisp meets this requirements is provided in [67, §C]. Informally, these properties are:

1. **Privacy.** The system remains ϵ-differentially private for a given ϵ, or else everyone learns, with high probability, that the Aggregator cheated.
2. **Correctness.** When the Aggregator receives a response to a query, that response is correct – that is, the exact answer plus the noise required for ϵ-differential privacy.
3. **Liveness.** As long as there is sufficient privacy budget left, the Aggregator will continue to be able to query the system and receive responses with high probability.
4. **Indemnification.** If the Aggregator follows the protocol, devices cannot fabricate evidence that would prove that the Aggregator had deviated from the protocol.

4 Implementation

In this section, we give a quick overview of the Honeycrisp prototype we used for our experimental evaluation. The code is available under an open-source license at `https://github.com/danxinnoble/honeycrisp`.

Shamir secret sharing: We use the error-correction properties of Shamir sharing [69] to tolerate the possibility that after key generation, some committee members' devices go offline before the second MPC protocol. Thus the output of the key-generation protocol MPC is a Shamir sharing of the secret key among the k committee members such that any subset of size $t+1$ can reconstruct the secret, and such that no t nodes can learn anything (in an information-theoretic sense) about the secret. Shamir sharing also has the property that if there are at least $t+1$ honest nodes, the honest nodes can detect any errors introduced by dishonest nodes.

MPC: Our implementation focused on the major computational bottlenecks for our systems – the two MPC protocols. We implemented the MPC protocols using the SCALE-MAMBA framework [49]. SCALE-MAMBA is a compiler and virtual machine for running generic MPC computations. It is the successor to the SPDZ framework [23], and is based on many of the same protocols. SCALE-MAMBA is very well suited for our application: it is truly multiparty (able to compute an MPC between any number of parties), it is secure against malicious adversaries who deviate from the protocol, and it allows developers to express functions using familiar high-level programming syntax rather than boolean or arithmetic circuits.

Because SCALE-MAMBA provides Shamir-sharing as one of its built-in MPC sharing schemes, we were able to use this native scheme to store the secret key between the key generation and decryption rounds. We modified the open-source SCALE-MAMBA source code to reconstruct the secret key automatically using existing shares, even if some of the nodes went offline between the key generation and the decryption.

SCALE-MAMBA operations are performed in a finite field modulo a prime p. This complements our Ring-LWE encryption scheme particularly well, since we could use p as the integer modulus for the LWE scheme. This meant that native SCALE-MAMBA operations were automatically modulo p, so we did not need to implement the modular arithmetic within the MPC. Furthermore, SCALE-MAMBA allows this prime modulus to be configured. In Ring-LWE, the additive homomorphism of plaintexts is modulo some integer q, where $|p \mod q| \ll q$, ideally $p = 1 \mod q$. Being able to specify p allowed us to have a sufficiently large plaintext modulus to hold the aggregation.

Ring-LWE: Honeycrisp requires an additively homomorphic cryptosystem to aggregate user inputs, and we instantiate our scheme using the simple "two-element" Ring-LWE-based encryption scheme of [54]. We chose this encryption scheme because its key generation and decryption operations are very simple algebraically – each involves a small constant number of additions and one multiplication in the ring $\mathbb{Z}_p[x]/(x^n + 1)$ where p is prime and n is a power of 2.

The encryption scheme works over a polynomial ring $R_p \overset{\text{def}}{=} \mathbb{Z}_p[x]/(x^n + 1)$. Then the secret key is a random polynomial $s(x) \in R_p$, and the public key is a pair generated by sampling a random $a \in R_p$ and setting the public key to be $(a, b) \in R_p^2$, where $b \overset{\text{def}}{=} a \cdot s + e \in R_p$, for some "error" $e \in R_p$ chosen from an appropriate error distribution. The plaintext space is $\mathbb{Z}_q^l$, where $q, l \in \mathbb{Z}$, $l \leq n$, $q \ll p$ and $|p \mod q| \ll q$. To encrypt a vector $z \in \mathbb{Z}_q^l$, the encryptor generates a random $r \in R_p$, and computes the ciphertext $(u, v) \overset{\text{def}}{=} (a \cdot r + e_1, b \cdot r + \lfloor p/q \rceil \cdot z) \in R_p^2$. Decryption is then simply $z = round(v - u \cdot s, \lfloor p/q \rceil)/\lfloor p/q \rceil$ where $round(x, y)$ rounds each coefficient of x to the nearest multiple of y. (This assumes the errors e, e_1, e_2 are sufficiently small relative to p/q).

Our design and implementation for Ring-LWE key generation and decryption inside of an MPC was developed independently from the concurrent work of [47], except that we use their observation that if the plaintext length, l, is less than n, then only l coefficients of v ever need to be stored.

Security parameters: We use the LWE-estimator tool [53] of Albrecht et al. [2] to obtain concrete parameters that provide sufficiently high security based on the best current LWE attack algorithms. Using this tool, we find that dimensionality $n = 4096$, a 128-bit prime p, and a Gaussian error distribution with $\sigma = \frac{\sqrt{2}}{2}$ (which we approximate as the centered binomial distribution with $N = 2$ trials) in each dimension, provides over 128 bits of security.

We note that there is a space-time tradeoff: on the one hand, Ring-LWE's easy decryption and key generation simplify the committee's MPCs, and the large dimension allows many metrics to be aggregated in parallel – while our implementation only uses one counter, our choices can yield up to 4,096 counters, each with a capacity of about 50 bits! But on the other hand, the ciphertexts are fairly large, which increases the bandwidth cost of the aggregator (Section 5.5). With a different homomorphic encryption scheme, such as Paillier [61] or elliptic-curve-based El Gamal (ECEG), the MPCs would take longer, but the ciphertexts would be smaller.

The verification portion of our scheme requires a collision resistant hash function (for the Merkle Trees) and a signature scheme for each user. Following standard practice, we use a SHA-256 hash function and RSA-2048 signatures.

5 Evaluation

Our goal for the experimental evaluation is to answer the following three questions: 1) Can Honeycrisp support periodic queries while giving reasonable privacy guarantees?, 2) How expensive is Honeycrisp in terms of computation, bandwidth, and storage?, and 3) How well does Honeycrisp scale?

5.1 Experimental setup

Honeycrisp is designed to operate in a very large deployment with potentially billions of laptops and phones, as well as a large data center. Since we did not have access to a large enough testbed, we benchmarked several of the components individually. For user-side computations, this is safe, since users communicate only with the aggregator and not with each other, and for the aggregator's computations we can easily extrapolate the cost because the operations are simple and can mostly be done in parallel. The only component of Honeycrisp that requires more attention is the committee; here, we cannot simply extrapolate, but fortunately the committees are small enough for us to run the corresponding computations completely.

Our aggregator experiments were run on eight PowerEdge R430 servers with 64 GB of RAM, two Xeon E5-2620 CPUs, and 10 Gbps Ethernet. The operating system was Fedora Core 26 with a Linux 4.3.15 kernel. This equipment seems reasonably close to what a real-world aggregator would have in its data center. To simulate users operating in a global setting, we used multiple `t2.large` Amazon EC2 servers with 8 GB of RAM, located in all available geographic regions, to obtain realistic latencies and communication costs.

We compare three different systems: 1) a RAPPOR-style solution (LDP) that achieves differential privacy in the local setting by making use of the randomized response mechanism; 2) a hypothetical solution (GDP) that uploads the unencrypted data to the aggregator, which then releases the result using global differential privacy, but not the SVT, somewhat analogous to PINQ [55]; and 3) our proposed solution, Honeycrisp. Notice that the second solution cannot protect user privacy against the aggregator, so it is not necessarily a realistic comparison point for Honeycrisp; nonetheless we demonstrate an improvement over this generic setting.

5.2 Utility

Our first goal is to determine whether Honeycrisp really can support queries for longer than existing systems. To this end, we simulate a comparison over a 10-year span between LDP, GDP, and Honeycrisp. We consider a simple vector of sensitivity-1 counting queries, which is at the heart of the count-mean sketches Apple is using [8], and we assume that the query needs to be asked once per day. Our model query is performed on a corpus of Twitter data spanning 5 years, and the count-mean sketch is over word usage frequency for newly-appearing words in the English language. We assume $N = 1.3 \cdot 10^9$ users, which was the size of Apple's deployment in February 2018 [4], and we choose the parameters in such a way that the total, noised count is within 1% of the true count with probability $p = 0.95$, assuming a query with a constant fraction that .001% of users respond to (although this error is a constant factor that affects all systems identically). For Honeycrisp, we set a threshold of 5%, and we (conservatively) assume that, on average, the true count changes by that amount about once every three months. This seems realistic: research on changing use of out-of-vocabulary language, as well as our own queries, show that word frequency changes as little as $1 - 1.5\%$ over an entire year [30].

Figure 2 shows a simulation of the privacy budget consumption of all three systems over time. The LDP-based system has the highest consumption by far; it goes through a budget of $\varepsilon = 1$ (a common choice [43], indicated by the horizontal line) approximately every 91 days. This is because, in the local setting, each user's data must be noised individually, so the sum contains much more noise than with global differential privacy, where the sum is computed precisely and then noised only once. As discussed in [28], with n this results in an incurred error cost of $O(\sqrt{n})$, as opposed to $O(1)$ in

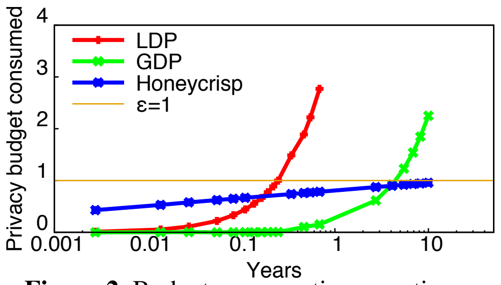


Figure 2. Budget consumption over time.

the global setting. This is consistent with Apple's decision to renew the privacy budget very frequently, and if more users were to respond to every query, this cost would become even higher! The consumption of the (insecure and hypothetical) GDP-based system is lower, but a budget of $\varepsilon = 1$ would last less than half as long as Honeycrisp over this time span, when we consider both systems operating over data vectors of size 10. This is because Honeycrisp has a second advantage: with the sparse-vector technique, the budget decreases logarithmically with the total number of queries (as opposed to linearly) and needs to be charged substantially only in the case when the answer changes. Because of this, Honeycrisp can run for 10 years without exhausting its privacy budget of $\varepsilon = 1$. For additional details, please see [67, §A].

Note that at first, the bound on the privacy budget for Honeycrisp is *higher* than that of both the LDP and GDP. This is because of the way the SVT works: it charges a relatively large privacy cost at the beginning, based on the expected number of times the data will change, and then charges only logarithmically for queries where the analyst's estimate turns out to be approximately correct. (The cost for such queries is not exactly zero because the threshold comparison is performed on the already-noised answer, so there will be occasional charges even when the estimate is correct.) In contrast, the other systems' privacy budgets degrade linearly, so, in the long run, Honeycrisp uses its privacy budget much more efficiently.

Even at this much lower rate of consumption for Honeycrisp, any finite privacy budget will eventually run out. However, "recharging" the privacy budget is not unreasonable per se, since many secrets would become far less valuable to an adversary if it took years to learn them. The key question is how frequently the budget needs to be recharged, and here Honeycrisp outperforms basic randomized response in the local setting by a factor of over 40.

5.3 Cost: Normal participants

Next, we examine the cost that a "normal" participant would pay to be part of Honeycrisp. We measured these costs by running all the participant-level steps in a single round of the protocol; we report the storage, bandwidth, and computation time for five system sizes: $N = 1.3 \cdot 10^9$ (the estimated size of Apple's deployment), as well as, for comparison, values ranging from $N = 1.3 \cdot 10^8$ to $N = 1.3 \cdot 10^{10}$.

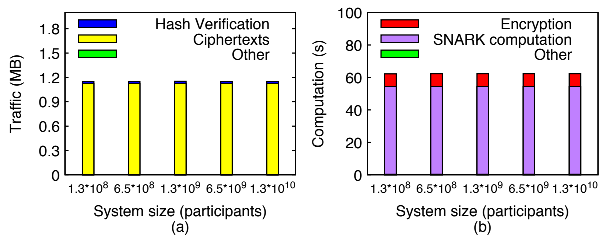


Figure 3. Bandwidth (a) and computation (b) required of each participant in each round.

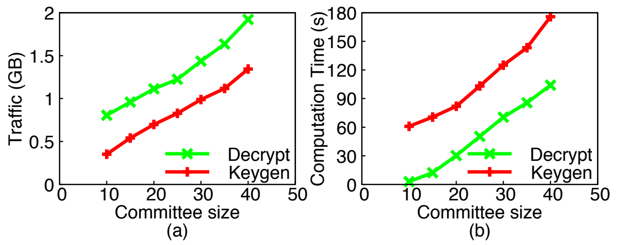


Figure 4. Bandwidth (a) and time (b) required for each of the two MPC steps using SCALE-MAMBA, for a RLWE cryptosystem in a lattice of dimension $n = 4096$.

Bandwidth: Figure 3(a) shows the amount of bandwidth that is consumed in a single round. The amount grows slightly with the system size because the MHT becomes taller and thus its inclusion proofs become longer (with $O(\log N)$). However, at less than 1.2 MB, the overall amount is reasonable even for the largest system size we tried. The commitments and range proofs (in particular, zk-SNARKs) each require less than 1 kB [10], which is too little to be visible in the figure.

Computation: Figure 3(b) shows the amount of computation that a participant needs to perform in each round, in terms of milliseconds of computation time on an E5-2620 core. Checking the signatures on the certificate and the MHT inclusion proof consumes only a small amount of time; the overall amount is likely dominated by the prover's computation. The implementation of [10] has proof times of approximately 0.2 ms per arithmetic gate. Considering the size of the arithmetic circuit implementing our RLWE encryption scheme, this would result in a proving time of approximately 54 seconds. Although this cost is high, each device would need to perform this step only once per query. At one query per day, this should be manageable, especially if (as in our motivating scenarios) quick turnaround times are not required and the computation can be done slowly in the background.

Storage: Participant machines do not need to permanently store any information, since they can always download the entire history of blocks and Merkle-tree roots from the bulletin board. However, it makes sense to store at least the most recent randomness block, most recent certificate, and at most 3 ciphertexts at a time for summation verification, which together would be less than 200 kB.

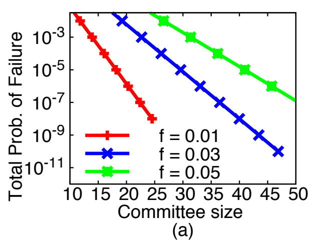
5.4 Cost: Committee

We now quantify the cost of a participant that has been chosen as a committee member for the current round. Such a participant must perform two additional steps: 1) the MPC to generate the keypair, as discussed in Section 3.3, and 2) the MPC to decrypt, noise, and threshold the aggregate, as discussed in Section 3.6. (There are other small costs, such as signing the certificate, but we ignore them here because the MPC costs clearly dominate.) These costs are independent of the number N of participants, but they do very much depend on the committee size, which is why we vary this parameter from 10 to 40 users.

In Figure 4(a), we show the total number of bytes that are sent by a committee member in each of the two MPCs; Figure 4(b) shows the total completion time for each MPC. We see that both time and traffic scale linearly with the size of the committee. With $C = 40$ committee members, each committee member uses less than 5 minutes and about 3.3 GB for both protocols combined. The MPC execution consists of an online phase and a secure pre-processing phase that generates randomness. The latter is responsible for much of the cost, making it difficult (but not impossible) to run this process on mobile devices.

If the cost is too high for the mobile devices, there are at least two possible solutions. One is to avoid mobile devices entirely and to ask only more powerful devices (laptops or desktops) to serve on the committee. If the adversary cannot target specific device types, this merely results in a smaller pool of potential committee members. If the adversary *can* target the candidate devices specifically, this approach would require us to scale down the fraction f of malicious devices; for instance, if we assume $f = 3\%$ but two thirds of the devices are mobile, we would need to choose the other parameters based on $f = 1\%$ instead. The other way is to leverage a party with limited trust, if one happens to be available. We do not discuss this option here due to lack of space, but in [67, §B], we show that it can reduce the cost to almost zero. Our experiments with the SPDZ multiparty compiler show that, in this case, the online phase alone requires just 10 MB for both protocols combined.

Next, we justify our choice of committee sizes. Figure 5(a) shows the probability of a privacy failure during a 10-year period, given various settings for the fraction of malicious nodes f and the committee size C. With $f = 3\%$ and a committee of $C = 40$ members, the chance of *ever* seeing a privacy failure (that is, a committee with too many malicious nodes) during the ten years is about 10^{-8}. Figure 5(b) similarly shows, for various settings of f and the fraction of offline nodes g, the minimum committee size that would be needed to ensure that at least 95% of the queries receive an answer. Again, with $f = 3\%$ and $g = 4\%$, a committee of $C = 40$ members would be sufficient. Notice that, if more nodes are offline than the choice of g anticipates, the result is simply that a few more queries will go unanswered.



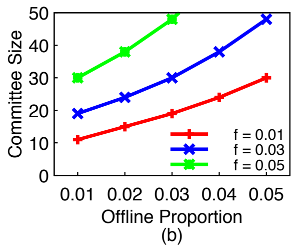
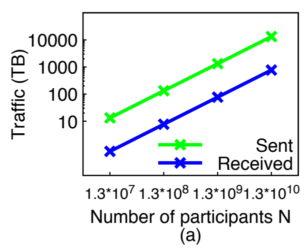


Figure 5. a) Probability of privacy failure for various committee sizes; b) minimum committee size needed for liveness.



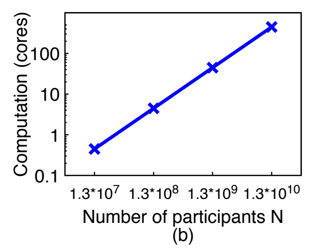


Figure 6. Bandwidth (a) and computation (b) required for the aggregator.

5.5 Cost: Aggregator

Finally, we turn our attention to the aggregator. The aggregator clearly has the highest workload, but it also presumably has the most resources. Since we cannot fully replicate the aggregator in our lab, we benchmark the various steps individually and then extrapolate. As before, we focus on an estimated size of $N = 1.3 \cdot 10^9$, as well as, for comparison, values ranging from $N = 1.3 \cdot 10^7$ to $N = 1.3 \cdot 10^{10}$.

Bandwidth: The aggregator would need to receive, from each client, a public key and a ciphertext. (We ignore the single copy of the certificate and the final result that the aggregator receives from the committee because they are insignificant.) The aggregator would need to send, to each client, a MHT inclusion proof, a copy of the committee's certificate, and a selection of ciphertexts for summation tree verification, using $s = 5$, giving 99% verification of correctness (see Section 3.5), and thus requiring at most 17 ciphertexts to be sent to each user. The only variable-size items are the inclusion proofs, which require $N \log N$ bytes given N participants, and the number of ciphertexts, which scales linearly; the public keys are 256 bytes each, the ciphertexts 65,552 bytes each, and the certificates 92 bytes each using an RSA certificate.

Figure 6(a) shows the total amount of bandwidth (bytes sent or received) that the aggregator would need in each round. Overall, the bandwidth consumption grows with $O(N \log N)$, with a strong linear component. At $N = 1.3 \cdot 10^9$, the amount sent would be roughly 1450 TB, or 1.12 MB/user; for comparison, this would be less than the amount of traffic generated by having 60% of the users download a typical web page (about 2 MB [17]) from the aggregator. If the traffic is a concern, it could be reduced to about 1.45 TB by using ECEG instead of Ring-LWE, at the expense of somewhat longer MPCs for the committee, as discussed in Section 4.

Computation: The aggregator would need to generate the MHT, verify the range proof that each participant uploads, and perform the homomorphic addition. (The inclusion proofs do not require extra work because they can simply be read from the MHT once it is generated.) With our choices for the hash function (SHA-256) and the homomorphic cryptosystem (Ring-LWE), a single hash operation takes 0.005 ms and a single homomorphic addition takes 1.7 ms. For the range proofs, we estimate a verification cost of 5 ms, based on [10].

Figure 6(b) shows the total computation cost in terms of computation time on a single E5-2620 core. If we (somewhat arbitrarily) require the computation phase of each round to last no more than an hour, the aggregator would need 45 cores for $N = 1.3 \cdot 10^9$, which seems achievable.

Storage: The aggregator would need to store the public keys and ciphertexts of all the participants and the MHT. (The range proofs can be discarded once verified.) With 2048-bit keys, SHA-256 hashes, and LWE encryption, a public key, a single hash, and a ciphertext consume 256 bytes, 32 bytes, and 65,552 bytes respectively, so the overall storage requirement is 65.84 kB per user, or roughly 86 TB for $N = 1.3 \cdot 10^9$. Again, this seems clearly within the power of a typical aggregator.

6 Related Work

Honeycrisp offers three key properties: global differential privacy, absence of powerful trusted parties, and scalability to billions of users. To our knowledge, no existing system has achieved all three properties simultaneously.

Local differential privacy: Several distributed differentially private systems add noise locally to each user's input, instead of once to the final result. This avoids the need for expensive cryptography, but it requires more noise, and thus reduces accuracy. One prominent example of such a system is RAPPOR [32, 33]. The schemes of [1, 38] also require participants to add noise locally, however, rather than use homomorphic encryption to hide the users' inputs from the aggregate, they use pair-wise blinding factors. Additional theoretical contributions have also operated in the local setting, but have included additional cryptographic tools [19, 46, 70]. More recent theoretical work tackles similar challenges of infrequently changing *data* – [45] assumes that the data is drawn from some set of underlying distributions that aren't changing, while [31] assumes that individuals' data aren't changing very frequently – by contrast Honeycrisp only makes assumptions about the changing nature of *answers to queries*.

A related approach is used in federated learning; for instance, [11, 14] update complex models, such as neural networks, locally on user devices, to avoid sending data to a centralized aggregator. As with the earlier approaches, the accuracy gain comes at the expense of privacy.

Smaller-scale systems: Bindschaedler et al. [12] considers differentially private aggregation with an untrusted aggregator and a strict star topology (users never communicate with each other). Their system uses both local and global noise addition to provide security against collusion between the aggregator and users. However, [12] requires each user to perform $O(n)$ public-key encryptions and requires $O(n)$ communication between each user and the aggregator, and thus is not suitable in our setting, where n is very large, possibly up to a billion.

Halevi et al. [40] shows how to compute an *arbitrary* function in a setting where there are n users and single server. [40] does not guarantee differential privacy, but the full version of the paper does outline a system for securely computing a sum in this model [41][§ 4.3]. However, the users connect to the server *sequentially*, and each interaction with the server requires $\Omega(n)$ communication, which limits scalability.

Powerful trusted parties: Some existing systems rely on a trusted party – an assumption Honeycrisp avoids. For instance, PDDP [20] and PROCHLO [13] both make use of a proxy to send information from a client to an analyst, [24] uses a trusted third party, and [52] relies on a trusted dealer to set up keys. [64] does not rely on trust, but operates in a different setting where users communicate directly with each other.

Another group of prior solutions relies on the *anytrust* model, that is, a *group* of third parties that must include at least one honest party in order to protect privacy. The key differences to Honeycrisp are that these parties are static, which makes them easier targets for the adversary, and that they must each contribute substantial resources, which increases the difficulty of recruiting such parties. (In contrast, Honeycrisp uses dynamic committees and performs most of the expensive work at the aggregator, which is untrusted and has a clear incentive to contribute.) One example of such a system is UnLynx [36], which uses a group of trusted servers to help with shuffling, aggregation, and query processing. UnLynx supports richer queries than Honeycrisp (e.g., a SQL-style `GROUPBY`), but the servers' workload grows linearly with the data size, so, with a billion users, each server would have to be quite powerful. Outis [21] similarly supports GDP without a trusted party but requires two non-colluding semi-honest servers, with their cryptographic server serving as an analogue to Honeycrisp's committee functionality.

Prio [22] is another example from this group that also relies on a group of special servers for aggregation. As with UnLynx, each server needs substantial CPU and bandwidth resources. Like [56], Prio does not provide differential privacy; rather, it focuses on robustness to malicious user inputs, which it recognizes using a new kind of zero-knowledge proof. This makes Prio vulnerable, e.g., to intersection attacks, in which an analyst performs two identical queries but forces one device to be offline during the second query, so that its sensitive data can be computed from the two results. [22] does sketch a possible extension to add differential privacy; however, even with this addition, Prio's proof processing means that it is not as scalable to massive user bases as Honeycrisp – Prio can process around 300 submissions per second (which remains roughly constant no matter the number of servers), so it would take over 35 days to process a billion submissions.

Computing other functions: We note that there are several other systems that offer differential privacy while computing functions over distributed data, such as database joins [60] or vertex programs [62]. In each case, the underlying technology is quite specific to the class of functions that is being targeted: for instance, the core of [60] is a primitive for set-intersection cardinality, which has no obvious connection to the aggregations that Honeycrisp can perform.

Other privacy guarantees: Many existing systems for collecting sensitive data rely on secret-sharing [18, 44, 50], anonymizing networks [35, 51] or even systems like Tor [42, 63] to aggregate the data privately, but do not explicitly make use of differential privacy.

Bonawitz et al. [15] considers a scenario that is similar to ours, and presents a protocol that also offers strong protection against user drop-out during protocol runs. It uses pair-wise blinding to hide user inputs (as in [1, 38]), but does not focus on differential privacy. Instead, the server learns the *exact* summation, but only if a certain threshold of inputs are received. This approach requires pair-wise key exchange between all parties (and thus $\Omega(n^2)$ communication); scalability is achieved by performing the aggregation in many small batches of n values (in the evaluation, $n \leq 500$). Since the threshold is less than n, the anonymity set is on the order of hundreds, even if there are millions of users.

7 Conclusion

Honeycrisp fills a gap in the space of secure aggregation systems: it can stretch a given privacy budget much longer – possibly over as much as ten years – as long as the underlying data does not change too often, and it does so in a highly scalable way, without introducing a trusted party. Thus, Honeycrisp could help to address the criticism of existing deployments, e.g., the one operated by Apple, by addressing the unique threat model that these data aggregators face. Honeycrisp does require a nontrivial amount of computation from the (small) group of user devices that is serving on the committee, but the recent improvements in MPC implementations (e.g., [3, 16, 73]) make it seem likely that this cost can be further reduced in the coming years.

Acknowledgments

We thank our shepherd, James Mickens, and the anonymous reviewers for their thoughtful comments and suggestions. Aaron Roth provided helpful feedback on earlier versions of this paper. This work was supported in part by ONR grant N00014-15-1-2750 and by NSF grants CNS-1703936, CNS-1563873, CNS-1513671, and CNS-1513694.

References

[1] G. Ács and C. Castelluccia. I have a DREAM! (DiffeRentially privatE smArt Metering). In *Proc. International Workshop on Information Hiding*, 2011.

[2] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptography*, 9:169–203, 2015.

[3] A. Aly, M. Keller, D. Rotaru, P. Scholl, N. P. Smart, and T. Wood. SCALE-MAMBA software. `https://homes.esat.kuleuven.be/~nsmart/SCALE/`, 2018.

[4] Apple. Apple reports first quarter results. Press release, February 2018; `https://www.apple.com/newsroom/2018/02/apple-reports-first-quarter-results/`.

[5] Apple. Differential privacy. `https://images.apple.com/privacy/docs/Differential_Privacy_Overview.pdf`.

[6] Apple. Reports on government information requests. `https://www.apple.com/legal/transparency/report-pdf.html`.

[7] Apple. A message to our customers. `https://www.apple.com/customer-letter/`, Feb. 2016.

[8] Apple Differential Privacy Team. Learning with privacy at scale. *Apple Machine Learning Journal*, 1(8), Dec. 2017.

[9] E. K. B. Laurie, A. Langley. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[10] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a Von Neumann architecture. In *Proc. USENIX Security*, 2014.

[11] 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]*, Dec. 2018.

[12] V. Bindschaedler, S. Rane, A. E. Brito, V. Rao, and E. Uzun. Achieving differential privacy in secure multiparty data aggregation protocols on star networks. In *Proc. CODASPY*, Mar. 2017.

[13] 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 *Proc. SOSP*, Oct. 2017.

[14] 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 *Proc. SysML*, 2019.

[15] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proc. CCS*, 2017.

[16] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. *IACR Cryptology ePrint Archive*, 2019:448, 2019.

[17] C. Buckler. Average page weight increases 15% in 2014, Dec. 2014. `https://www.sitepoint.com/average-page-weight-increases-15-2014/`.

[18] C. Castelluccia, A. C.-F. Chan, E. Mykletun, and G. Tsudik. Efficient and provably secure aggregation of encrypted data in wireless sensor networks. *ACM Trans. Sen. Netw.*, 5(3):20:1–20:36, June 2009.

[19] T.-H. H. Chan, E. Shi, and D. Song. Private and Continual Release of Statistics. *ACM Trans. Inf. Syst. Secur.*, 14(3):26:1–26:24, Nov. 2011.

[20] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *Proc. NSDI*, 2012.

[21] A. R. Chowdhury, C. Wang, X. He, A. Machanavajjhala, and S. Jha. Outis: Crypto-Assisted Differential Privacy on Untrusted Servers. *arXiv:1902.07756 [cs]*, Feb. 2019.

[22] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proc. NSDI*, Mar. 2017.

[23] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proc. CRYPTO*. 2012.

[24] G. Danezis, C. Fournet, M. Kohlweiss, and S. Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proc. SEGS*, 2013.

[25] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Proc. CRYPTO*, 1989.

[26] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proc. TCC*, 2006.

[27] C. Dwork, M. Naor, O. Reingold, G. N. Rothblum, and S. Vadhan. On the complexity of differentially private data release: Efficient algorithms and hardness results. In *Proc. STOC*, 2009.

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

[29] C. Dwork, G. N. Rothblum, and S. Vadhan. Boosting and differential privacy. In *Proc. FOCS*, 2010.

[30] J. Eisenstein. What to do about bad language on the internet. *NAACL-HLT*, 2013(4):359–369, June 2013.

[31] U. Erlingsson, V. Feldman, I. Mironov, A. Raghunathan, K. Talwar, and A. Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *Proc. SODA*, 2019.

[32] U. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proc. CCS*, 2014.

[33] 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]*, Mar. 2015.

[34] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *Proc. USENIX ATC*, 2005.

[35] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proc. CCS*, 2002.

[36] 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 *Proc. PETS*, Oct. 2017.

[37] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proc. SOSP*, 2017.

[38] S. Goryczka and L. Xiong. A comprehensive comparison of multiparty secure additions with differential privacy. *IEEE Transactions on Dependable and Secure Computing*, 14(5):463–477, 2017.

[39] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proc. USENIX Security*, Aug. 2011.

[40] S. Halevi, Y. Lindell, and B. Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *Proc. CRYPTO*, 2011.

[41] S. Halevi, Y. Lindell, and B. Pinkas. Secure computation on the web: Computing without simultaneous interaction. IACR ePrint 2011/157, 2011.

[42] S. Hohenberger, S. Myers, R. Pass, and a. shelat. ANONIZE: A large-scale anonymous survey system. In *Proc. IEEE S&P*, 2014.

[43] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *Proc. CSF*, July 2014.

[44] M. Jawurek and F. Kerschbaum. Fault-tolerant privacy-preserving statistics. In *Proc. PETS*, 2012.

[45] M. Joseph, A. Roth, J. Ullman, and B. Waggoner. Local Differential Privacy for Evolving Data. In *Proc. NIPS*, 2018.

[46] M. Joye and B. Libert. A Scalable Scheme for Privacy-Preserving Aggregation of Time-Series Data. In *Proc. FC*, 2013.

[47] M. Kraitsberg, Y. Lindell, V. Osheter, N. P. Smart, and Y. T. Alaoui. Adding distributed decryption and key generation to a ring-lwe based cca encryption scheme. In *Australasian Conference on Information Security and Privacy*, pages 192–210. Springer, 2019.

[48] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proc. USENIX Security*, 2012.

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

[50] K. Kursawe, G. Danezis, and M. Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *Proc. PETS*, 2011.

[51] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis. Towards efficient traffic-analysis resistant anonymity networks. *SIGCOMM Comput. Commun. Rev.*, 43(4):303–314, Aug. 2013.

[52] I. Leontiadis, K. Elkhiyaoui, M. Önen, and R. Molva. PUDA - Privacy and unforgeability for data aggregation. In *Proc. CANS*, 2015.

[53] LWE estimator tool. `https://bitbucket.org/malb/lwe-estimator/`, commit 3019847.

[54] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Proc. EUROCRYPT*, 2010.

[55] F. McSherry. Privacy integrated queries. In *Proc. SIGMOD*, 2009.

[56] L. Melis, G. Danezis, and E. De Cristofaro. Efficient Private Statistics with Succinct Sketches. In *Proc. NDSS*, 2016.

[57] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Proc. CRYPTO*, Berlin, Heidelberg, 1987.

[58] I. Mironov. On significance of the least significant bits for differential privacy. In *Proc. CCS*, 2012.

[59] I. Morris. Apple Has Sold 1.2 Billion iPhones Worth $738 Billion In 10 Years. *Forbes*. June 29, 2017. `https://www.forbes.com/sites/ianmorris/2017/06/29/apple-has-sold-1-2-billion-iphones-worth-738-billion-in-10-years/`.

[60] A. Narayan and A. Haeberlen. DJoin: Differentially private join queries over distributed databases. In *Proc. OSDI*, 2012.

[61] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT*, 1999.

[62] A. Papadimitriou, A. Narayan, and A. Haeberlen. DStress: Efficient differentially private computations on distributed data. In *Proc. EuroSys*, 2017.

[63] R. A. Popa, A. J. Blumberg, H. Balakrishnan, and F. H. Li. Privacy and accountability for location-based aggregate statistics. In *Proc. CCS*, 2011.

[64] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proc. SIGMOD*, 2010.

[65] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM Journal*, 8(2):300–304, 1960.

[66] A. Roth and T. Roughgarden. Interactive privacy via the median mechanism. In *Proc. STOC*, 2010.

[67] E. Roth, D. Noble, B. H. Falk, and A. Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core (extended version). Technical Report MS-CIS-19-03, University of Pennsylvania, Sept. 2019.

[68] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proc. NSDI*, 2010.

[69] A. Shamir. How to share a secret. *CACM*, 22(11):612–613, 1979.

[70] E. Shi, H. Chan, E. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *Proc. NDSS*, 2011.

[71] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. Scalable bias-resistant distributed randomness. In *Proc. IEEE S&P*, 2017.

[72] J. Tang, A. Korolova, X. Bai, X. Wang, and X. Wang. Privacy loss in Apple's implementation of differential privacy on MacOS 10.12, 2017. `https://arxiv.org/pdf/1709.02753.pdf`.

[73] X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. In *Proc. CCS*, 2017.

[74] T. Wolverton. iPhone sales crater 15% in Apple's worst holiday results in a decade, and the forecast looks just as grim. *Business Insider*, Jan 2019. `https://www.businessinsider.com/apple-q1-2019-earnings-iphone-sales-revenue-eps-analysis-2019-1`.

[75] A. Yao. Protocols for secure computations (extended abstract). In *Proc. FOCS*, 1982.