Privacy and Accountability for Location-based Aggregate Statistics

Raluca Ada Popa
MIT
ralucap@mit.edu

Andrew J. Blumberg
University of Texas, Austin
blumberg@math.utexas.edu

Hari Balakrishnan
MIT
hari@mit.edu

Frank H. Li
MIT
frankli@mit.edu

ABSTRACT

A significant and growing class of location-based mobile applications aggregate position data from individual devices at a server and compute aggregate statistics over these position streams. Because these devices can be linked to the movement of individuals, there is significant danger that the aggregate computation will violate the *location privacy* of individuals. This paper develops and evaluates PrivStats, a system for computing aggregate statistics over location data that simultaneously achieves two properties: first, provable guarantees on location privacy even in the face of any side information about users known to the server, and second, privacy-preserving accountability (i.e., protection against abusive clients uploading large amounts of spurious data). PrivStats achieves these properties using a new protocol for uploading and aggregating data anonymously as well as an efficient zero-knowledge proof of knowledge protocol we developed from scratch for accountability. We implemented our system on Nexus One smartphones and commodity servers. Our experimental results demonstrate that PrivStats is a practical system: computing a common aggregate (e.g., count) over the data of $10,000$ clients takes less than 0.46 s at the server and the protocol has modest latency (0.6 s) to upload data from a Nexus phone. We also validated our protocols on real driver traces from the CarTel project.

Categories and subject descriptors: C.2.0 [Computer Communication Networks]: General–*Security and protection*
General terms: Security

1. INTRODUCTION

The emergence of location-based mobile services and the interest in using them in road transportation, participatory sensing [30, 22], and various social mobile crowdsourcing applications has led to a fertile area of research and commercial activity. At the core of many of these applications are mobile nodes (smartphones, in-car devices, etc.) equipped with GPS or other position sensors, which (periodically) upload time and location coordinates to a server. This information is then processed by the server to compute a variety of aggregate statistics.

As primary motivation, consider applications that process streams of GPS position and/or speed samples along vehicle trajectories, sourced from smartphones, to determine current traffic statistics such as average speed, average delay on a road segment, or congestion at an intersection. Several research projects (e.g., CarTel [22], Mobile Millennium [29]) and commercial products (e.g., TomTom [37]) provide such services. Another motivation is social mobile crowdsourcing, for example, estimating the number of people at a restaurant to determine availability by aggregating data from position samples provided by smartphones carried by participating users.

A significant concern about existing implementations of these services is the violation of user *location privacy*. Even though the service only needs to compute an aggregate (such as a mean, standard deviation, density of users, etc.), most implementations simply continuously record time-location pairs of all the clients and deliver them to the server, labeled by which client they belong to [22, 29, 23]. In such a system, the server can piece together all the locations and times belonging to a particular client and obtain the client's path, violating her location privacy.

Location privacy concerns are important to address because many users perceive them to be significant (and may refuse to use or even oppose a service) and because they may threaten *personal security*. Recently (as of the time of writing this paper), two users have sued Google [28] over location data that Android phones collect citing as one of the concerns "serious risk of privacy invasions, including stalking." The lawsuit attempts to prevent Google from selling phones with software that can track user location. Just a week before, two users sued Apple [26] for violating privacy laws by keeping a log of user locations without offering users a way to disable this tracking or delete the log. Users of TomTom, a satellite navigation system, have expressed concern over the fact that TomTom logs user paths and sells aggregate statistics (such as speeding hotspots) to the police, who in turn install speed cameras [37]. A study by Riley [38] shows even wider location privacy fears: a significant number of drivers in the San Francisco Bay Area will not install toll transponders in their cars because of privacy concerns. Moreover, online databases are routinely broken into or abused by insiders with access; if that happens, detailed records of user mobility may become known to criminals, who can then attempt security attacks, such as burglarizing homes when they know that their residents are away [43].

In this paper, we design, implement, and evaluate **PrivStats**, a practical system for computing aggregate statistics in the context of mobile, location-based applications that achieves both strong guarantees of location privacy and protection against cheating clients. PrivStats solves two major problems: it provides formal location privacy guarantees against *any side information* (SI) attacks, and it provides client *accountability without a trusted party*. Because of these contributions, in comparison to previous systems [17], Clique-Cloak [15], [20], [25], [21], and Triplines [19], PrivStats provides the strongest formal privacy and correctness guarantees while making the weakest trust assumptions.

Side information refers to any out-of-bound information the server may have, which, when used together with the data the server gets in a system, can help the server compromise user location privacy. Some previous work ensures that clients upload data free of identifiers (they upload approximate time, location, and data sample), but even when all data is anonymized, a considerable amount of private location information can still leak due to side information. As a simple example, if the server knows that Alice is the only person living on a street, it can infer that Alice just left or arrived at her house when receiving some speed updates from that street. SI can come in many forms and has been shown to leak as much as full client paths in some cases: Krumm [25], as well as Gruteser and Hoh [18], show that one can infer driver paths from anonymized data and then link them to driver identities using side information such as knowledge of map, typical driving behavior patterns, timing between uploads, and a public web service with addresses and names. Despite SI's ability to leak considerable location privacy, previous systems for mobile systems aggregates either do not address the problem at all, or they only treat specific cases of it, such as areas of low density (see §11).

Our first contribution is to provide *an aggregate statistics protocol with strong location privacy guarantees, including protection against any general side information attack.* We formalize our guarantees by providing a definition of location privacy, termed *strict location privacy* or SLP (Def. 2), which specifies that the server learns nothing further about the location of clients in the face of arbitrary side information other than the desired aggregate result; then, in §5, we provide a protocol, also termed SLP, that provably achieves our definition.

While we manage to hide most sources of privacy leakage in our SLP protocol without any trust assumptions, hiding the number of tuples generated by clients information-theoretically can be reduced, under reasonable model assumptions, to a problem in distributed algorithms that can be shown to be impossible to solve. The reason is that it requires synchronization among a set of clients that do not know of each other. Our solution is to use a lightweight and restricted module, the *smoothing module* (SM), that helps clients synchronize with respect to the number of tuples upload and performs one decryption per aggregate. We distribute the SM on the clients (each "client SM" is responsible for handling a few aggregates) so as to ensure that at most a small fraction of SMs misbehave. We provide our strong SLP guarantees for all aggregates with honest SMs, while still ensuring a high level of protection for compromised SMs:

- A compromised SM cannot change aggregate results undetectably and cannot collude with clients to allow them to corrupt aggregates.
- Although a malicious SM opens the doors to potential SI leakage from the number of uploads and the values uploaded, a significant degree of privacy remains because client uploads are always anonymized, free of timing and origin information. Our SM is distributed *on the clients*, resulting in few compromised client SMs in practice.
- Our design of the SM (§5) facilitates distribution and ensuring its correctness: the SM has light load, ≈ 50 times less load than the server, performs two simple tasks summing up to only 62 lines of code (excluding libraries) which can be easily scrutinized for correctness, and only uses constant storage per aggregate.
- In contrast to our limited SM, trusted parties in previous work [19, 15, 21, 17] had the ability to fully compromise client paths and modify aggregate results when corrupted; at the same time, their use still did not provide general SI attack protection.

Since clients are anonymous, the problem of accountability becomes serious: malicious clients can significantly affect the correctness of the aggregate results. Hoh et al. [20] present a variety of attacks clients may use to change the aggregate result. For example, clients might try to divert traffic away from a road to reduce a particular driver's travel time or to keep low traffic in front of one's residence, or divert traffic toward a particular road-way to increase revenue at a particular store. A particular challenge is that accountability and privacy goals are in tension: if each client upload is anonymous, it seems that the client could upload as many times as she wishes. Indeed, most previous systems [17, 15, 25, 21] do not provide any protection for such client biasing attacks and the few that provide such verification ([19, 20]) place this task on heavily loaded trusted parties that when compromised can release full location paths with user identities and can corrupt aggregate results.

Our second contribution is *a privacy-preserving accountability protocol without any trusted parties* (§6). We provide a cryptographic protocol for ensuring that each client can upload at most a fixed quota of values for each aggregate. At the core of the protocol, is an efficient zero-knowledge proof of knowledge (ZKPoK) that we designed from scratch for this protocol. The zero-knowledge property is key in maintaining the anonymity of clients. Therefore, no trusted party is needed; in particular, the SM is not involved in this protocol.

Finally, our third contribution is an implementation of the overall system on Nexus One smartphones and commodity servers (§9). One of our main goals was to design a practical system, so we strived to keep our cryptographic protocols practical, including designing our ZKPoK from scratch. Computing a common aggregate (e.g. count) over the data of 10^4 clients takes less than 0.46 s at the server, the protocol has modest latency of about 0.6 s to upload data from a Nexus phone, and the throughput is linearly scalable in the number of processing cores. We believe these measurements indicate that PrivStats is practical. We also validated that our protocols introduce little or no error in statistics computation using real driver traces from the CarTel project [22].

2. MODEL

In this section, we discuss the model for PrivStats. Our setup captures a wide range of aggregate statistics problems for mobile systems.

2.1 Setting

In our model, we have many clients, a server, and a smoothing module (SM). *Clients* are mobile nodes equipped with smartphones: drivers in a vehicular network or peers in a social crowdsourcing application. The *server* is an entity interested in computing certain aggregate statistics over data from clients. The SM is a third party involved in the protocol that is distributed on clients.

We assume that clients communicate with the server and the SM using an anonymization network (e.g., Tor), a proxy forwarder, or other anonymizing protocol to ensure that privacy is not compromised by the underlying networking protocol. In this paper, we assume these systems succeed in hiding the origin of a packet from an adversary; it is out of our scope to ensure this. We assume that these systems also hide network travel time; otherwise clients must introduce an appropriate delay so that all communications have roughly the same round trip times.

Clients can choose which aggregates they want to participate in; they can opt-out if the result is deemed too revealing.

A *sample* is a client's contribution to an aggregate, such as average speed, delay on a road, or a bit indicating presence in a certain restaurant. Clients can generate samples periodically, as is the case in vehicular systems, or at certain location/times, as is the case for social mobile crowdsourcing systems. A *sample point* is a pair consisting of a location and a time interval where/when clients should generate a sample. We say that a client *passes through a sample point* if the client passes by the location and in the time interval of a sample point.

An aggregate to be computed consists of a sample point and the type of data clients should sample. The server may only be interested in computing certain aggregates. We assume that clients know the aggregates of interest (e.g., from a public database or a deterministic algorithm), that is, at which sample points they should upload and what kind of samples they should generate. We assume that each aggregate has an identifier id, with the mapping between identifiers and aggregates publicly known (e.g., a hash). Therefore, clients generate tuples of the form $\langle \mathsf{id}, \mathsf{sample} \rangle$, meaning that their sample for aggregate id is sample. We denote by *aggregate result* the result of the aggregation of all client samples for an aggregate. Each aggregate has an associated *sample interval* denoting an interval of acceptable values for the sample. For example, suppose that $\mathsf{id} = 2$ corresponds to an aggregate with sample point road segment S and time interval between T_1 = 4.00 pm to T_2 = 4.15 pm, and type of sample "average speed" (the sample interval could be 0 mph – 100 mph). A client passing through S, in the interval of time $[T_1, T_2]$ should take a sample of their average speed. The client generates the tuple: $\langle 2, 24 \text{ mph} \rangle$. Finally, the aggregate result over all clients could be 30 mph. (Note that the server may choose to compute the function in "real time" or at some later point in time.)

To preserve privacy, clients in PrivStats will transform these tuples (e.g., encrypt and add cryptographic tokens) and use a certain upload strategy to upload the transformed tuples to the server. Note that clients are not uploading their identity with the tuples, unlike in some existing systems [22, 13].

2.2 Threat Model

The server *cannot be trusted to protect the privacy of clients*: the server might attempt to recover the path of a given client and release this information to third parties without a client's consent or knowledge (e.g., for advertising), or hackers could break into the server and steal location data about particular users. These attacks can use any side information. However, the server is trusted to compute the aggregate statistics correctly: this is its goal.

Clients are also untrusted: they may attempt to bias the aggregate result (e.g., to convince a server that the road next to their house is crowded and no more traffic should come). There are two such potential threats: uploading a large number of samples or uploading samples with out-of-range samples, both of which could change an aggregate result significantly. However, each client is allowed to upload within a server-imposed quota and an acceptable interval of sample values; checking that the sample uploaded is precisely correct is out-of-scope. (See §10.1 for more details).

The smoothing module (SM) can misbehave by trying to change aggregate results or colluding with clients to help them bias statistics; PrivStats prevents such behavior. To guarantee full side information protection, the requirement on the SM is that it does not leak timing or decrypted samples.

3. SIDE INFORMATION AND LOCATION PRIVACY

Side information can leak information about client paths even when the server receives tuples without identities. As already discussed, the number of tuples in areas of low density can leak privacy. Krumm [25] as well as Gruteser and Hoh [18] inferred driver paths and their identities using SI such as knowledge of map, driving patterns, upload timing, and a public web service. Driving patterns

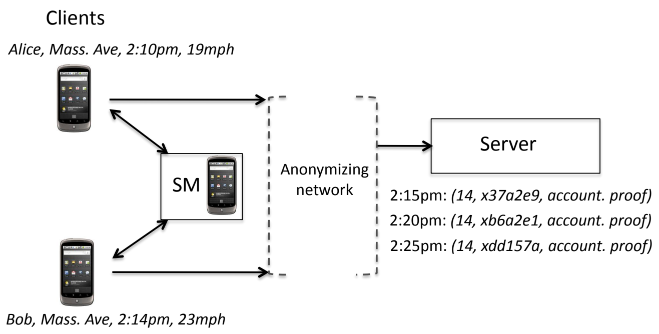

Clients
Alice, Mass. Ave, 2:10pm, 19mph
SM
Anonymizing network
Server
2:15pm: (14, x37a2e9, account. proof)
2:20pm: (14, xb6a2e1, account. proof)
2:25pm: (14, xdd157a, account. proof)
Bob, Mass. Ave, 2:14pm, 23mph

Figure 1: Architecture of PrivStats.

coupled with sample values can also leak private information. For example, if the server knows that client Bob is a habitual speeder, tracking uploads with high speed values will likely indicate his path. Other two interesting examples of SI are physical observation and information about someone else's path. For an example of the latter, assume that the server knows that Alice and Bob both went on street A and then it sees one upload from street B1 and one upload from B2, which follow directly from A. Knowing that Bob's house is on B1, it can infer that Alice went on B2. To quantify how much privacy is leaked, Shokri et al. [41] offer a framework for quantifying privacy leakage in various location privacy schemes that also takes into account side information.

We now provide a definition of location privacy that is resilient to arbitrary side information. SI can come in different forms and reveal varying degrees of information, making it challenging to provide theoretical guarantees. For example, one can never guarantee that the path of a client between point A and point B will remain unknown because the client may simply be observed physically at some location. Moreover, one cannot prevent the adversary from having SI: such SI can come from a variety of out-of-bound sources (public databases, physical observation, collusion with clients, personal knowledge of a driver's trends, etc.). Instead, the idea is that the protocol should not reveal any *additional* information about client paths beyond what the server already knows and the desired aggregate result. In short, the protocol should reveal nothing else other than the aggregate result.

Consider the following example. Alice, Bob, and some other clients volunteer to participate in an average speed computation on street S_1 for time interval 7 am to 9 am. Now suppose that the server has the following side information about the clients: on the days that Alice goes to work, she leaves home on street S_1 at 8:15 am; also, Bob is a speeder and tends to drive significantly faster than the average driver. If the protocol satisfies our definition, the server will learn the average speed, say 30 mph, and nothing else. In particular, the server will not learn how many people passed through the street or whether there was an upload at 8:15 am (and hence whether Alice went to work). Moreover, the server will not see the individual speed values uploaded so it cannot determine if there were some high speeds corresponding to Bob.

For a security parameter k and an aggregate function F (which can also be a collection of aggregate functions), consider a protocol $\mathbb{P} = \mathbb{P}^F(k)$ for computing F. For example, the SLP protocol (§5) is such a protocol $\mathbb{P}$ where F can be any of the aggregates discussed in §7. Let R be a collection of raw tuples generated by users in some time period; that is, R is a set of tuples of the form $\langle \mathsf{id}, \mathsf{sample} \rangle$ together with the precise time, client network information, and client id when they are generated. R is therefore the private data containing the paths of all users and should be hidden from the

I. System setup (Runs once, when the system starts).

1: Server generates public and private keys for accountability by running *System setup* from §6.
2: Both Server and SM generate public and private keys for the aggregation protocols by running *System setup* from §5.

II. Client join (Runs once per client, when a client signs up to the system).

1: Client identifies herself and obtains public keys and capabilities to upload data for accountability (by running *Client join*, §6) from Server and public keys for aggregation (by running *Client join*, §5) from SM. Server also informs Client of the aggregates Server wants to compute, and Client decides to which aggregates she wants to contribute (i.e., generate tuples).

III. Client uploads for aggregate id (Runs when a client generates a tuple $\langle \mathsf{id}, \mathsf{sample}\rangle$)

1: Client runs SLP's *Upload* protocol (§5) by communicating with the SM for each $\langle \mathsf{id}, \mathsf{sample}\rangle$ and produces a set of transformed tuples $\mathcal{T}_1, \ldots, \mathcal{T}_k$ and a set of times, $t_1, \ldots, t_k$, when each should be uploaded to Server.
2: For each i, Client uploads $\mathcal{T}_i$ to Server at time t_i.
3: Client also proves to Server using the *Accountability upload* protocol, §6, that the upload is within the permissible quotas and acceptable sample interval.

IV. Server computes aggregate result id (Runs at the end of an aggregate's time interval).

1: Server puts together all the tuples for the same id and aggregates them by running *Aggregation* with SM, §5.

Figure 2: Overview of PrivStats.

server. We refer to R as a *raw-tuple configuration*. Let SI be some side information available at the server about R. Using $\mathbb{P}$, clients transform the tuples in R before uploading them to the server. (The sizes of R, F, and SI are assumed to be polynomial in k.)

The following definition characterizes the information available at the server when running protocol $\mathbb{P}$. Since we consider a real system, the server observes the timing and network origin of the packets it receives; a privacy definition should take these into account.

DEF. 1 (SERVER'S VIEW.). *The view of the server,* $\mathsf{View}_{\mathbb{P}}(R)$, *is all the tuples the server receives from clients or SM in* $\mathbb{P}$ *associated with time of receipt and any packet network information, when clients generate raw tuples* R *and run protocol* $\mathbb{P}$.

Let $\mathcal{D}$ be the domain for all raw-tuple configurations R. Let res be some aggregate result. Let $\mathcal{R}^F(\mathsf{res}) = \{R \in \mathcal{D} : F(R) = \mathsf{res}\}$ be all possible collections of raw tuples where the associated aggregate result is res.

Security game. We describe the SLP definition using a game between a challenger and an adversary. The adversary is a party that would get access to all the information the server gets. Consider a protocol $\mathbb{P}$, a security parameter k, a raw-tuple domain $\mathcal{D}$, and an adversary Adv.

1: The challenger sets up $\mathbb{P}$ by choosing any secret and public keys required by $\mathbb{P}$ with security parameter k and sends the public keys to Adv.
2: Adv chooses SI, res, and $R_0, R_1 \in \mathcal{R}^F(\mathsf{res})$ (to facilitate guessing based on SI) and sends R_0 and R_1 to the challenger.
3: Challenger runs $\mathbb{P}$ producing $\mathsf{View}^{\mathbb{P}}(R_0)$ and $\mathsf{View}^{\mathbb{P}}(R_1)$, and sends them to Adv. It chooses a fair random bit b and sends $\mathsf{View}^{\mathbb{P}}(R_b)$ to Adv. ($\mathsf{View}^{\mathbb{P}}$ usually contains probabilistic encryption so two encryptions of R_b will lead to different values.)
4: The adversary outputs its best guess for b^* and wins this game if $b = b^*$. Let $\mathsf{win}^{\mathbb{P}}(\mathsf{Adv}, k) := \Pr[b = b^*]$ be the probability that Adv wins this game.

DEF. 2 (STRICT LOCATION PRIVACY – SLP). *A protocol* $\mathbb{P}$ *has strict location privacy with respect to a raw-tuple domain,* $\mathcal{D}$*, if, for any polynomial-time adversary* Adv, $\mathsf{win}^{\mathbb{P}}(\mathsf{Adv}, k) \leq 1/2 +$ *negligible function of* k.

Intuitively, this definition says that, when the server examines the data it receives and any side information, all possible configurations of client paths having the same aggregate result are equally likely. Therefore, the server learns nothing new beyond the aggregate result.

Strict location privacy and differential privacy. The guarantee of strict location privacy is complementary to those of differential privacy [12], and these two approaches address different models. In SLP, the server is untrusted and all that it learns is the aggregate result. In a differential privacy setting, the *server is trusted and knows all private information of the clients*, clients issue queries to the database at the server, and clients only learn aggregate results that do not reveal individual tuples. Of course, allowing the server to know all clients' private path information is unacceptable in our setting. PrivStats' model takes the first and most important step for privacy: hiding the actual paths of the users from the server. Actual paths leak much more than common aggregate results in practice. A natural question is whether one can add differential privacy on top of PrivStats to reduce leakage from the aggregate result, while retaining the guarantees of PrivStats, which we discuss in §8.

4. OVERVIEW

PrivStats consists of running an aggregation protocol and an accountability protocol. The aggregation protocol (§5) achieves our strict location privacy (SLP) definition, Def. 2, and leaks virtually nothing about the clients other than the aggregate result. The accountability protocol (§6) enables the server to check three properties of each client's upload *without* learning anything about the identity of the client: the client did not exceed a server-imposed quota of uploads for each sample point, did not exceed a total quota of uploads over all sample points, and the sample uploaded is in an acceptable interval of values.

Figure 1 illustrates the interaction of the three components in our system on an example: computation of aggregate statistic with id 14, average speed. The figure shows Alice and Bob generating data when passing through the corresponding sample point and then contacting the SM to know how many tuples to upload. As we explain in §5, we can see that the data that reaches the server is anonymized, contains encrypted speeds, arrives at random times independent of the time of generation, and is accompanied by accountability proofs to prevent malicious uploads. Moreover, the number of tuples arriving is uncorrelated with the real number.

Figure 2 provides an overview of our protocols, with components elaborated in the upcoming sections.

5. AGGREGATION: THE SLP (STRICT LOCATION PRIVACY) PROTOCOL

A protocol with strict location privacy must hide all five leakage vectors (included in Def. 1): client identifier, network origin of packet, time of upload, sample value, and number of tuples generated for each aggregate (i.e., the number of clients passing by a sample point). The need to hide the first two is evident and we already discussed the last three vectors in §3.

Hiding the identifier and network origin. As discussed in §2, clients never upload their identities (which is possible due to our accountability protocol described in §6). We hide the network origin using an anonymizing network as discussed in §2.1.

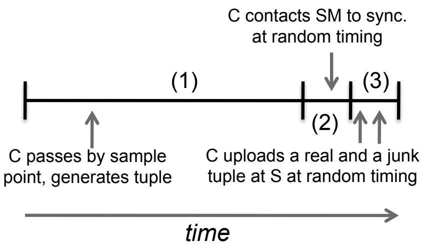
Hiding the sample. Clients encrypt their samples using a (semantically secure) homomorphic encryption scheme. Various schemes can be used, depending on the aggregate to be computed; Paillier [32] is our running example and it already enables most common aggregates. Paillier has the property that $E(a) \cdot E(b) = E(a + b)$, where $E(a)$ denotes encryption of a under the public key, and the multiplication and addition are performed in appropriate groups. Using this setup, the server computes the desired aggregate on encrypted data; the only decrypted value the server sees is the final aggregate result (due to the SM, as described later in this section).

It is important for the encryption scheme to be *verifiable* to prevent the SM from corrupting the decrypted aggregate result. That is, given the public key PK and a ciphertext $E(a)$, when given a and some randomness r, the server can verify that $E(a)$ was indeed an encryption of a using r, and there is no b such that $E(a)$ could have been an encryption of b for some randomness. Also, the holder of the secret key should be able to compute r efficiently. Fortunately, Paillier has this property because it is a trapdoor permutation; computing r involves one exponentiation [32].

Hiding the number of tuples. The server needs to receive a number of tuples that is independent of the actual number of tuples generated. The idea is to arrange that the clients will *upload in total a constant number of tuples*, U_{id}, for each aggregate id. U_{id} is a publicly-known value, usually an upper bound on the number of clients generating tuples, and computed based on historical estimates of how many clients participate in the aggregate. §9 explains how to choose U_{id}.

The difficulty with uploading U_{id} in total is that clients do not know how many other clients pass through the same sample point, and any synchronization point may itself become a point of leakage for the number of clients.

Assuming that there are no trusted parties (i.e., everyone tries to learn the number of tuples generated), under a reasonable formalization of our specific practical setting, the problem of hiding the number of tuples can be reduced to a simple distributed algorithms problem which is shown to be impossible (as we show in Appendix A). For this reason, we use a party called the *smoothing module* (SM) that clients use to synchronize and upload a constant number of tuples at each aggregate. The SM will also perform the final decryption of the aggregate result. The trust requirement on the SM is that it does not decrypt more than one value for the server per aggregate and it does not leak the actual number of tuples. Although the impossibility result provides evidence that some form of trust is needed for SLP, we do not claim that hiding the number of tuples would remain impossible if one weakened the privacy guarantees or altered the model. However, we think our trust assumption is reasonable, as we explain in §8: a malicious SM has limited effect on privacy (it does not see the timing, origin, and identifier of client tuples) and virtually no effect on aggregate results (it cannot decrypt an incorrect value or affect accountability). Moreover, the SM has light load, and can be distributed on clients ensuring that most aggregates have our full guarantees. For simplicity, we describe the SLP protocol with one SM and subsequently explain how to distribute it on clients.



C contacts SM to sync. at random timing
(1)
(2)
(3)
C passes by sample point, generates tuple
C uploads a real and a junk tuple at S at random timing
time

Figure 3: **Staged randomized timing upload for a client C: (1) is the generation interval, (2) the synchronization interval, and (3) the upload interval.**

Since the number of real tuples generated may be less than U_{id}, we will arrange to have clients occasionally upload *junk* tuples: tuples that are indistinguishable from legitimate tuples because the encryption scheme is probabilistic. Using the SM, clients can figure out how many junk tuples they should upload. The value of the junk tuples is a neutral value for the aggregate to be computed (e.g., zero for summations), as we explain in §7.

To summarize, the SM needs to perform only two simple tasks:

- Keep track of the number of tuples that have been uploaded so far (s_{id}) for an aggregate (without leaking this number to an adversary). Clients use this information to figure out how many more tuples they should upload at the server to reach a total of U_{id}.
- At the end of the time period for an aggregate id, decrypt the value of the aggregate result from the server.

Hiding timing. Clients upload tuples according to a staged randomized timing algorithm described in Figure 3; this protocol hides the tuple generation time from both the server and the SM. The generation interval corresponds to the interval of aggregation when clients generate tuples, and the synchronization and upload interval are added at the end of the generation interval, but are much shorter (they are not points only to ensure reasonable load per second at the SM and server).

We now put together all these components and specify the SLP protocol. Denote an aggregate by id and let quota be the number of samples a client can upload at id. Let $[t_{s0}, t_{s1}]$ be the sync. interval.

I. System setup: SM generates a Paillier secret and public key.
II. Client joins: Client obtains the public key from the SM.
III. Upload:

```
   ○ GENERATION INTERVAL
1: Client generates a tuple $\langle \mathsf{id}, \mathsf{sample} \rangle$ and computes $\mathcal{T}_1 :=$
   $\langle \mathsf{id}, E[\mathsf{sample}] \rangle$ using SM's public key.
2: Client selects a random time $t_s$ in the sync. interval and does
   nothing until $t_s$.
   ○ SYNCHRONIZATION INTERVAL
```

```
3: Client ↔ SM : At time $t_s$, Client requests $s_{\mathsf{id}}$, the number
   of tuples other clients already engaged to upload for id from
   SM ($s_{\mathsf{id}} = 0$ if Client is the first to ask $s_{\mathsf{id}}$ for id).
4: Client should upload $\Delta s := \min(\mathsf{quota}, \lceil U_{\mathsf{id}}(t_s - t_{s0})/(t_{s1} - t_{s0}) \rceil - s_{\mathsf{id}})$ tuples. If $\Delta s \leq 0$, but $s_{\mathsf{id}} < U_{\mathsf{id}}$, as-
   sign $\Delta s := 1$. (In parallel, SM updates $s_{\mathsf{id}}$ to $s_{\mathsf{id}} := \Delta s + s_{\mathsf{id}}$
   because it also knows quota and it assumes that Client will
   upload her share.)
5: Client picks $\Delta s$ random times, $t_1, \ldots, t_{\Delta s}$, in the upload
   interval to use when uploading tuples.
   ◦ UPLOAD INTERVAL:
6: Client → Server : The client uploads $\Delta s$ tuples at times
   $t_1, \ldots, t_{\Delta s}$. One of the tuples is the real tuple $\mathcal{T}_1$, the others,
   if any, are junk tuples.

IV. Aggregation:
1: At the end of the interval, Server computes the aggregate on
   encrypted samples by homomorphically adding all samples
   for id.
2: Server ↔ SM : Server asks SM to decrypt the aggregate re-
   sult. SM must decrypt only one value per aggregate. Server
   verifies that SM returned the correct decrypted value as dis-
   cussed.
```

The equation for Δs is designed so that for each time t in the synchronization interval, approximately a total of $U_{\mathsf{id}}(t - t_{s0})/(t_{s1} - t_{s0})$ tuples have been scheduled by clients to be uploaded during the upload interval. Therefore, when t becomes t_{s1}, U_{id} tuples should have been scheduled for upload.

We assign $s_{\mathsf{id}} := 1$ when Δs comes out negative or zero because as long as $s_{\mathsf{id}} < U_{\mathsf{id}}$, it is better to schedule clients for uploading in order to reach the value of U_{id} by the end of the time interval; this approach avoids too many junk tuples (which do not carry useful information) being uploaded later in the interval if insufficiently many clients show up then. Moreover, this ensures that, as long as U_{id} is an upper bound on the number of clients passing by, few clients will refrain from uploading in practice.

Choice of U_{id} and quota. A reasonable choice for quota is 3 and $U_{\mathsf{id}} = \mathsf{avg}_{\mathsf{id}} + \mathsf{std}_{\mathsf{id}}$, where $\mathsf{avg}_{\mathsf{id}}$ is an estimate of the average of clients passing through a sample point and $\mathsf{std}_{\mathsf{id}}$ of the standard deviation; both these values can be deduced based on historical records or known trends of the sample point id. We justify and evaluate this choice of U_{id} and quota in §10.1.

Let $\mathcal{D}^*$ be the set of all raw-tuple configurations in which clients succeed in uploading U_{id} at sample point id. Raw-tuple configurations in practice should be a subset of this set for properly chosen U_{id} and quota.

THEOREM 1. *The SLP protocol achieves the strict location privacy definition from Def. 2 for $\mathcal{D}^*$.*

PROOF. According to Def. 2, we need to argue that the information in the view of the server (besides the aggregate result and any SI the server already knows) when clients generate two different collections of raw tuples, R_0 and R_1, is indistinguishable. The information that reaches the server for id is a set of tuples consisting of three components: time of receipt, network origin, and encrypted sample. Due to our U_{id} scheme, the number of tuples arriving at the server in both cases is the same, U_{id}. Next, our staged upload protocol ensures that the timing of upload is uniformly at random distributed and the network origin is hidden. The encryption scheme of the samples is semantically-secure so, by definition, any two samples encrypted with this scheme are indistinguishable to any polynomial adversary. Since all these three components are independent of each other, the total set of tuples for R_0 and R_1 are indistinguishable. □

Note that this theorem assumes that the clients, server, and SM run the SLP protocol correctly; specifically, the server performs all tasks in the protocol correctly, although it may try to infer private information from the data it receives. In §8, we discuss the protection PrivStats provides when these parties deviate from the SLP protocol in various ways.

Note that we provide the SLP guarantees as long as clients succeed in uploading U_{id} tuples (as captured in the definition of $\mathcal{D}^*$). In the special cases when an aggregate is usually popular (so it has high U_{id}), but suddenly falls in popularity (at least $\mathsf{quota} = 3$ times less), clients may not be able to reach U_{id}. One potential solution is to have clients volunteer to "watch" over certain sample points they do not necessarily traverse, by uploading junk tuples within the same quota restrictions, if s_{id} is significantly lower than U_{id} towards the end of the synchronization interval.

Also note that the results of an aggregate statistics are available at the server only at the end of the generation, synchronization, and upload intervals. The last two intervals are short, but the first one is as long as the time interval for which the server wants to compute the aggregate. This approach inherently does not provide real-time statistics; however, it is possible to bring it closer to real-time statistics, if the server chooses the generation interval to be short, e.g., the server runs the aggregation for every 5 minutes of an hour of interest.

5.1 Distributing the Smoothing Module

So far, we referred to the smoothing module as one module for simplicity, but we recommend the SM be distributed on the clients: this distributes the already limited trust placed on the SM. Each client will be responsible for running the SM for one or a few aggregate statistics. We denote by a *client SM* the program running on a client performing the task of the SM for a subset of the sample points. Therefore, each physical client will be running the protocol for the client described above and the protocol for one client SM (these two roles being logically separated).

A set of K clients will handle each aggregate to ensure that the SLP protocol proceeds even if $K - 1$ clients become unavailable (e.g., fail, lose connectivity). Assuming system membership is public, anyone can compute which K clients are assigned to an aggregate id using consistent hashing [24]; this approach is a standard distributed systems technique with the property that, roughly, load is balanced among clients and clients are assigned to aggregates in a random way (see [24] for more details).

The design of the SM facilitates distribution to smartphones: the SM has light load ($\approx$ 50 times less than the server), performs two simple tasks, and uses constant storage per aggregate (see §9).

We now discuss how the the two tasks of a client SM – aggregate result decryption and maintaining the total number of tuples – are accomplished in a distributed setting.

During system setup, one of the K clients chooses a secret and a public key and shares them with the other $K - 1$ clients. The server can ask any SM to decrypt the aggregate result: since the server can verify the correctness of the decryption, the server can ask a different SM in case it establishes that the first one misbehaved. Client SMs should notify each other if the server asked for a decryption to prevent the server from asking different client SMs for different decryptions. Decryption is a light task for client devices because it happens only once per aggregate.

For a given aggregate, the aggregate result will be successfully decrypted if at least one of K client SMs is available. K should be chosen based on the probability that clients become unavailable, which depends on the application. Since it is enough for only one SM to function for correctness, in general we expect that a small number for K (e.g., $K = 4$ should suffice in practice).

For maintaining the total number of tuples, we designed our staged timing protocol (Fig. 3) in such a way that client SMs need only be available for the synchronization time interval. This interval is shorter than the generation interval. A client should ask each client SM corresponding to a sample point for the value of s_{id} in parallel. Since some client may not succeed in contacting all the client SMs, the client SMs should synchronize from time to time. To facilitate client SMs synchronization, each client should choose a random identifier for the request and provide it to each client SM so that SMs can track which requests they missed when they synchronize. (As an optimization, for very popular aggregates, one might consider not trying to hide the number of tuples because the leakage is less for such aggregates. For example, an aggregate can be conservatively considered popular if there are at least 1000 clients passing per hour.)

Our strict privacy guarantees will hold for an aggregate if none of the K client SMs for that aggregate are malicious. Since the fraction of malicious clients is small in practice, our strong privacy guarantees will be achieved for most of the aggregates.

6. ACCOUNTABILITY

Since each client uploads tuples anonymously, malicious clients might attempt to bias the aggregates by uploading multiple times. In this section, we discuss PrivStats's accountability protocol, which enforces three restrictions to protect against such attacks: a quota for each client's uploads for each sample point, a total upload quota per client, and a guarantee that each sample value must be within an acceptable interval. The protocol must also achieve the conflicting goal of protecting location privacy. The most challenging part is to ensure that each client uploads within quota at each sample point; the other two requirements are provided by previous work.

We note that the SM is not involved in any part of accountability; the server will perform the accountability checks by itself. This is one of the main contributions of PrivStats: with no reliance on a (even partially or distributed) trusted party, the server is able to enforce a quota on the number of uploads of each client without learning who performed any given upload.

The idea is to have each client upload a *cryptographic token* whose legitimacy the server can check and which does not reveal any information about the client; furthermore, a client cannot create more that her quota of legitimate tokens for a given aggregate. Thus, if a client exceeds her quota, she will have to re-use a token, and the server will note the duplication, discarding the tuple.

Notation and conventions. Let SK_s be the server's secret signing key and PK_s the corresponding public verification key. Consider an aggregate with identifier id. Let T_{id} be the token uploaded by a client for aggregate id. All the values in this protocol are computed in $\mathbb{Z}_n$, where $n = p_1 p_2$, two safe prime numbers, even if we do not always mark this fact. We make standard cryptographic assumptions that have been used in the literature such as the strong-RSA assumption.

6.1 Protocol

For clarity, we explain the protocol for a quota of one tuple per aggregate id per client and then explain how to use the protocol for a larger quota.

The accountability protocol consists of an efficient *zero-knowledge proof of knowledge* protocol we designed from scratch for this problem. Proofs of knowledge [16], [3], [39] are proofs by which one can prove that she knows some value that satisfies a certain relation. For example, Schnorr [39] provided a simple and efficient algorithm for proving possession of a discrete log. The zero-knowledge [16] property provides the guarantee that no information about the value in question is leaked.

System setup. The server generates a public key (PK_s) and a private key (SK_s) for the signature scheme from [10].

Client joins. Client identifies herself to Server (using her real identity) and Server gives Client one capability (allowing her to upload one tuple per aggregate) as follows. Client picks a random number s and obtains a blind signature from Server on s denoted $\mathsf{sig}(s)$ (using the scheme in [10]). The pair $(s, \mathsf{sig}(s))$ is a capability. A capability enables a client to create exactly one correct token T_{id} for each aggregate id. The purpose of $\mathsf{sig}(s)$ is to attest that the capability of the client is correct. Without the certification provided by the blind signature, Client can create her own capabilities and upload more than she is allowed.

The client keeps s and $\mathsf{sig}(s)$ secret. Since the signature is blind, Server never sees s or $\mathsf{sig}(s)$. Otherwise, the server could link the identity of Client to $\mathsf{sig}(s)$; it could then test each token T_{id} received to see if it was produced using $\mathsf{sig}(s)$ and know for which tuples Client uploaded and hence her path. By running the signing protocol only once with Client, Server can ensure that it gives only one capability to Client.

Accountability upload

1. Client $\to$ Server : Client computes $T_{\mathsf{id}} = \mathsf{id}^s \mod n$ and uploads it together with a tuple. Client also uploads a zero-knowledge proof of knowledge (ZKPoK, below in Sec. 6.2) with which Client proves that she knows a value s such that $\mathsf{id}^s = T_{\mathsf{id}} \mod n$ and for which she has a signature from the server.
2. Server $\to$ Client : Server checks the proof and discards the tuple if the proof fails or if the value T_{id} has been uploaded before for id (signaling an overupload).

We prove the security of the scheme in Sec. 6.2 and the appendix. Intuitively, T_{id} does not leak s because of hardness of the discrete log problem. The client's proof is a zero-knowledge proof of knowledge (ZKPoK) so it does not leak s either. Since T_{id} is computed deterministically, more than one upload for the same id will result in the same value of T_{id}. The server can detect these over-uploads and throw them away. The client cannot produce a different T^*_{id} for the same id by using a different s^* because she cannot forge a signature of the server for s^*; without such signature, Client cannot convince Server that she has a signature for the exponent of id in T^*_{id}. Here we assumed that id is randomly distributed.

Quotas. To enforce a $\mathsf{quota} > 1$, the server simply issues a quota of capabilities during registration. In addition, the server may want to tie quotas to aggregates. To do so, the server divides the aggregates into categories. For each category, the server runs system setup separately obtaining different SK_s and PK_s and then gives a different number of capabilities to clients for each category.

Quota on total uploads. We also have a simple cryptographic protocol to enforce a quota on how much a client can upload in total over all aggregates, not presented here due to space constraints; however, any existing e-cash scheme suffices here [8].

Ensuring reasonable values. As mentioned, in addition to quota enforcement, for each tuple uploaded, clients prove to the server that the samples encrypted are in an expected interval of values (e.g., for speed, the interval is $(0, 150)$) to prevent clients from uploading huge numbers that affect the aggregate result. Such proof is done using the efficient interval zero-knowledge proof of [5]; the server makes such interval publicly available for each aggregate.

We discuss the degree to which fake tuples can affect the result in section 10.1.

6.2 A Zero-Knowledge Proof of Knowledge

We require a commitment scheme (such as the one in [33]): recall that a commitment scheme allows a client Alice to commit to a value x by computing a ciphertext ciph and giving it to another client Bob. Bob cannot learn x from ciph. Later, Alice can open the commitment by providing x and a decommitment key that are checked by Bob. Alice cannot open the commitment for $x' \neq x$ and pass Bob's verification check.

Public inputs: id, T_{id}, PK_s, g, h
Client's input: s, $\sigma = \mathsf{sig}(s)$.
Server's input: SK_s

CONSTRUCTION 1. ***Proof that* Client *knows* s *and* σ *such that*** $\mathsf{id}^s = T$ ***and*** σ ***is a signature by* Server *on*** s**.**

1. Client *computes a Pedersen commitment to* s*:* $\mathsf{com} = g^s h^r \mod n$*, where* r *is random.* Client *proves to* Server *that she knows a signature* $\mathsf{sig}(s)$ *from the server on the value committed in* com *using the protocol in [10].*
2. Client *proves that she knows* s *and* r *such that* $T_{\mathsf{id}} = \mathsf{id}^s$ *and* $\mathsf{com} = g^s h^r$ *as follows:*
   (a) Client *picks* k_1 *and* k_2 *at random in* $\mathbb{Z}_n$*. She computes* $T_1 = g^{k_1} \mod n$*,* $T_2 = h^{k_2} \mod n$ *and* $T_3 = \mathsf{id}^{k_1} \mod n$ *and gives them to the server.*
   (b) Server *picks* c *a prime number, at random and sends it to* Client.
   (c) Client *computes* $r_1 = k_1 + sc$*,* $r_2 = k_2 + rc$ *and sends them to* Server.
   (d) Server *checks if* $\mathsf{com}^c T_1 T_2 \overset{?}{\equiv} g^{r_1} h^{r_2} \mod n$ *and* $T_{\mathsf{id}}^c T_3 \overset{?}{\equiv} \mathsf{id}^{r_1} \mod n$*. If the check succeeds, it outputs "ACCEPT"; else, it outputs "REJECT".*

This proof can be made non-interactive using the Fiat-Shamir heuristic [14] or following the proofs in [5]; due to space limits, we do not present the protocol here. The idea is that the client computes c herself using a hash of the values she sends to the server in a way that is computationally infeasible for the client to cheat.

THEOREM 2. *Under the strong RSA-assumption, Construction 1 is a zero-knowledge proof of knowledge that the client knows* s *and* σ *such that* $\mathsf{id}^s = T_{\mathsf{id}} \mod n$ *and* σ *is a signature by the server on* s*.*

The proof is at the end of this section, in Section 6.5.

6.3 Optimization

To avoid performing the accountability check for each tuple uploaded, the server can perform this check probabilistically; with a probability q, it checks each tuple for aggregate id. If the server notices an attempt to bias the aggregate (a proof failing or a duplicate token), it should then check all proofs for the aggregate from then on or, if it stored the old proofs it did not verify, it can check them all now to determine which to disregard from the computation. Note that the probability of detecting an attempt to violate the upload quote, Q, increases exponentially in the number of repeated uploads, n: $Q = 1 - (1-q)^n$. For $q = 20\%$, after 10 repeated uploads, the chance of detection is already 90%. We recommend using $q = 20\%$, although this value should be adjusted based on the expected number of clients uploading for an aggregate.

6.4 Generality of Accountability

The accountability protocol is general and not tied to our setting. It can be used to enable a server to prevent clients from uploading more than a quota for each of a set of "subjects" while preserving their anonymity. For example, a class of applications are online reviews and ratings in which the server does not need to be able to link uploads from the same user; e.g., course evaluations. Students would like to preserve their anonymity, while the system needs to prevent them from uploading more than once for each class.

6.5 Proof

To prove Theorem 2, we use standard zero knowledge proofs of knowledge definitions (specifically the ones in [27]). For our protocol, the Prover is the client and the Verifier is the server. The proof of the theorem uses the following lemma.

LEMMA 3. *Under the strong RSA assumption, Step 2 of Construction 1 is an honest-verifier zero-knowledge argument system with proof of knowledge property for the desired problem.*

PROOF. We need to prove three properties:
Completeness: If the client knows s and r, the server will accept. Let us verify that the checks in Step 2d succeed for an honest client. We have $\mathsf{com}^c T_1 T_2 = (g^s h^r)^c g^{k_1} h^{k_2} = g^{k_1+sc} h^{k_2+rc} = g^{r_1} h^{r_2}$ and $T^c T_3 = (\mathsf{id}^s)^c \mathsf{id}^{k_1} = \mathsf{id}^{k_1+sc} = \mathsf{id}^{r_1}$. Thus, for an honest client, the server will accept.

Proof of knowledge: We must provide an efficient knowledge extractor, K. Consider a prover P that makes the verifier accept with a non-negligible probability. Let K be a machine that performs the following:

1. Start a random instance of the Prover. Let it output T_1 and T_2.
2. Pick c_1 at random and send it to the prover. Receive r_{11} and r_{12} and see if checks in Step 2d verify. If the checks do not verify, start this algorithm from the beginning.
3. If the checks verify, rewind the Prover up to the point where it just outputs T_1 and T_2 (the same as above). Pick c_2 at random and send it to the Prover. Receive r_{21} and r_{22}. If the checks in Step 2d are not verified, start this algorithm from the beginning.
4. If $c_1 - c_2$ does not divide both $r_{11} - r_{21}$ and $r_{12} - r_{11}$, start this algorithm from the beginning.
5. If the checks verify, then output $s = \frac{r_{11}-r_{21}}{c_1-c_2}$ and $r = \frac{r_{12}-r_{22}}{c_1-c_2}$

Since the Prover has non-negligible probability of convincing the Verifier, we can see that the equality checks will succeed and we will execute Step 5 in expected polynomial time. [11] prove that Step 4 will happen in expected polynomial time, if the strong RSA assumption holds. Let us show that, in this case, s and r are indeed solutions to our problem.

Since the checks verify, it means that we have $\mathsf{com}^{c_1} T_1 T_2 = g^{r_{11}} h^{r_{12}}$, $T_{\mathsf{id}}^{c_1} T_3 = \mathsf{id}^{r_{11}}$, $\mathsf{com}^{c_2} T_1 T_2 = g^{r_{21}} h^{r_{22}}$, and $T_{\mathsf{id}}^{c_2} T_3 = \mathsf{id}^{r_{21}}$. By dividing the appropriate two equations, we can compute: $g^s h^r = g^{\frac{r_{11}-r_{21}}{c_1-c_2}} h^{\frac{r_{12}-r_{22}}{c_1-c)2}} = \left(g^{r_{11}-r_{21}} h^{r_{12}-r_{22}}\right)^{1/(c_1-c_2)} = \left(\mathsf{com}^{c_1-c_2}\right)^{1/(c_1-c_2)} = \mathsf{com}$

and $\mathsf{id}^s = \left(\mathsf{id}^{r_{11}-r_{21}}\right)^{1/(c_1-c_2)} = \left(T_{\mathsf{id}}^{c_1-c_2}\right)^{1/(c_1-c_2)} = T_{\mathsf{id}}$

Zero-knowledge: We must provide a simulator proof. We follow the proof in [35], which proves that the Schnorr [39] algorithm for proving knowledge of a discrete logarithm is zero-knowledge.

We construct a simulator that can generate a communication transcript with any verifier indistinguishable from a transcript between the prover and the verifier. The simulator performs the following. Picks c', k_1', r_1' and r_2' at random. Computes $T_1' = g^{k_1'}$, $T_2' \equiv g^{r_1'} h^{r_2'} \mathsf{com}^{-c'} g^{-k_1'} (\mod n)$ and $T_3' = \mathsf{id}^{r_1'} T_{\mathsf{id}}^{-c'}$. Sends T_1', T_2', T_3' to the verifier. Receives c from the verifier. If $c \neq c'$, restart the verifier and repeat this algorithm from the beginning. If

$c = c'$, send r'_1 and r'_2 to the verifier; output the transcript produced in this run.

Let us argue that this transcript $(T'_1, T'_2, T'_3, c', r'_1, r'_2)$ is indistinguishable from a transcript between the prover and the verifier. We drew c', k'_1, r'_1 and r'_2 at random and computed T'_1, T'_2, T'_3 based on these values. The distribution of these values is indistinguishable from choosing c', k'_1, T'_2, T'_3 at random and then considering random values of r'_1 and r'_2 that satisfy the verifier checks. The latter distribution is the same with the distribution in a real transcript.

This simulation has an expected polynomial time of success if the space of possible values for c is constrained to a polynomial-sized space. □

PROOF PROOF OF THEOREM 2. Camenisch and Lysyanskaya [10] show that the protocol in Step 1 is a zero-knowledge proof of knowledge. Lemma 3 shows that Step 2 is also a zero-knowledge proof of knowledge.

All we have to prove is that the client cannot run these two proving protocols for different values of s. We proceed by contradiction. Assume that the user proofs knowledge of s_1 according to the proof in Step 1 and knowledge of s_2 according to the proof in Step 2 such that $s_1 \neq s_2$. Let M be a machine that has access to the knowledge extractor for the first and second proofs. M can use these two extractors and obtain $s_1 \neq s_2$ and r_1, r_2 such that

$$C \equiv g^{r_1} h^{s_1} \equiv g^{r_2} h^{s_2} \mod n,$$

This means that

$$g^{r_1 - r_2} \equiv h^{s_1 - s_2} \mod n.$$

M can repeat this experiment for different randomness given to the extractors and obtain $s_1 - s_2$ which divides $r_1 - r_2$ in expected polynomial time. By dividing these values, M obtains the discrete log of h in base g and thus invert the discrete log problem. □

7. AGGREGATE STATISTICS SUPPORTED

SLP supports any aggregates for which efficient homomorphic encryption schemes and efficient verification or proof of decryption exist. Additive homomorphic encryption (e.g., Paillier [32]) already supports many aggregates used in practice, as we explain below. Note that if verification of decryption by the SM is not needed (e.g., in a distributed setting), a wider range of schemes are available. Also note that if the sample value is believed to not leak, arbitrary aggregates could be supported by simply not encrypting the sample.

Table 1 lists some representative functions of practical interest supported by PrivStats. Note the generality of sum and average of functions — F could include conditionals or arbitrary transformations based on the tuple. When computing the average of functions, some error is introduced due to the presence of junk tuples because each sample is weighted by the number of uploads. However, as we show in §10.1, the error is small (e.g., $< 3\%$ for average speed for CarTel).

Note in Table 1 that count is a special case and we can make some simplifications. We do not need the SM at all. The reason is that the result of the aggregate itself equals the number of tuples generated at the sample point, so there is not reason to hide this number any more. Also, since each sample of a client passing through the sample point is 1, there is no reason to hide this value, so we remove the sample encryptions. As such, there is no need for the SM any more. Instead of contacting the SM in the sync. interval (Fig. 3), clients directly upload to the server at the same random timing.

Additive homomorphic encryption does not support median, min, and max. For the median, one possible solution is to approximate it with the mean. One possible solution is to use order-preserving encryption [4]: for two values $a < b$, encryption of a will be smaller than encryption of b. This approach enables the server to directly compute median, min, and max. However, it has the drawback that the server learns the order on the values. An alternative that leaks less information is as follows: To compute the min, a client needs to upload an additional bit with their data. The bit is 1 if her sample is smaller than a certain threshold. The server can collect all values with the bit set and ask the SM to identify and decrypt the minimum. (Special purpose protocols for other statistics are similar.)

8. PRIVACY ANALYSIS

We saw that if the clients, smoothing module, and the server follow the SLP protocol, clients have strong location privacy guarantees. We now discuss the protection offered by PrivStats if the server, SM, or clients maliciously deviate from our strict location privacy protocol.

Malicious server. The server may attempt to ask the SM to decrypt the value of one sample as opposed to the aggregate result. However, this prevents the server from obtaining the aggregate result, since the SM will only decrypt once per aggregate id, and we assume that the server has incentives to compute the correct result. Nevertheless, to reduce such an attack, the SM could occasionally audit the server by asking the server to supply all samples it aggregated. The SM checks that there are roughly U_{id} samples provided and that their aggregation is indeed the supplied value.

The server may attempt to act as a client and ask the SM for s_{id} (because the upload is anonymous) to figure out the total number of tuples. Note that, if the aggregate is not underpopulated, if the server asks for s_{id} at time t, it will receive as answer approximately $U_{\mathsf{id}}(t - t_{s0})/(t_{s1} - t_{s0})$, a value it knew even without asking the SM. This is one main reason why we used the "growing s_{id}" idea for the sync. interval (Fig. 3), as opposed to just having the SM count how many clients plan to upload and then informing each client of the total number at the end of the sync. interval, so that they can scale up their uploads to reach U_{id}. Therefore, to learn useful information, the server must ask the SM for s_{id} frequently to catch changes in s_{id} caused by other clients. However, when the server asks the SM for s_{id}, the helper increases s_{id} as well. Therefore, the SM will reach the U_{id} quota early in the sync. interval and not allow other clients to upload; hence, the server will not get an accurate aggregate result, which is against its interest.

Malicious SM. Since the server verifies the decryption, the SM cannot change the result. Even if the SM colludes with clients, the SM has no effect on the accountability protocol because it is entirely enforced by the server. If the SM misbehaves, the damage is limited because clients upload tuples without identifiers, network origin, and our staged timing protocol for upload hides the time when the tuples were generated even from the SM. A compromised SM does permit SI attacks based on sample value and number of tuples; however, if the SM is distributed, we expect the fraction of aggregates with a compromised SM to be small and hence our SLP guarantees to hold for most aggregates.

The SM may attempt a DoS attack on an aggregate by telling clients that U_{id} tuples have already been uploaded and only allowing one client with a desired value to upload. However, the server can easily detect such cheating when getting few uploads.

Malicious clients. Our accountability protocol prevents clients from uploading too much at an aggregate, too much over all aggregates, and out-of-range values. Therefore, clients cannot affect the correctness of an aggregate result in this manner. As mentioned in §2, we do not check if the precise value a client uploads is correct, but we show in §10.1 that incorrect value in the allowable range likely will not introduce a significant error in the aggregate result.

Table 1: Table of aggregate statistics supported by PrivStats with example applications and implementation.

Aggregation	Example Application	Implementation
Summation	No. of people carpooled through an intersection.	Each client uploads encryption of the number of people in the car. Junk tuples are encryptions of zero.
Addition of functions $\sum F(\text{tuple}_i)$	Count of people exceeding speed limit. This is a generalization of summation.	Each client applies F to her tuple and uploads encryption of the resulting value. Junk tuples are encryptions of zero.
Average	Average speed or delay.	See average of functions where $F(\text{tuple}) = \text{sample}$.
Standard Deviation	Standard deviation of delays.	Compute average of functions (see below) where $F(\text{tuple}) = \text{sample}^2$ and denote the result Avg_1. Separately compute $F(\text{tuple}) = \text{sample}$ and denote the result Avg_2. The server computes $\sqrt{\mathsf{Avg}_2 - (\mathsf{Avg}_1)^2}$. This procedure also reveals average to the server, but likely one would also want to compute average when computing std. deviation.
Average of functions $\sum_{i=1}^{N} F(\text{tuple}_i)/N$	Average no. of people in a certain speed range. Average speed and delay. This is a generalization of average.	If count (see below) is also computed for the sample point, compute $\sum_{i=1}^{N} F(\text{tuple}_i)$ as above instead, and then the server can divide by the count. Otherwise, compute summation of functions, where junk tuples equal real tuples. The server divides the result by the corresponding U_{id}.
Count	Traffic congestion: no. of drivers at an intersection.	The SM is not needed at all for count computation. Clients do not upload any sample value (uploading simply $\langle \text{id} \rangle$ is enough to indicate passing through the sample point). There are no junk tuples.

Colluding clients may attempt to learn the private path of a specific other client. However, since our SLP definition models a general adversary with access to server data, such clients will not learn anything beyond the aggregate result and the SI they already knew (which includes, for example, their own paths). A client may try a DoS attack by repeatedly contacting the SM so that the SM thinks U_{id} tuples have been uploaded. This attack can be prevented by having the SM also run the check for the total number of tuples a client can upload; U_{id} for a popular aggregate should be larger than the number of aggregates a client can contribute to in a day. For a more precise, but expensive check, the SM could run a check for the quota of tuples per aggregate, as the server does.

Aggregate result. In some cases, the aggregate result itself may reveal something about clients' paths. However, our goal was to enable the server to know this result accurately, while not leaking additional information. As mentioned, clients can choose not to participate in certain aggregates (see §2).

Differential privacy protocols [12] add noise into the aggregate result to avoid leaking individual tuples; however, most such protocols leak all the private paths to the server by the nature of the model. §3 explains how the SLP and differential privacy models are complementary. A natural question is whether one can add differential privacy on top of the PrivStats protocol, while retaining PrivStats' guarantees. At a high level, the SM, upon decrypting the result, may decide how much noise to add to the result to achieve differential privacy, also using the number of true uploads it knows (the number of times it was contacted during the sync. interval). The design of a concrete protocol for this problem is future work. Related to this question is the work of Shi et al. [40], who proposed a protocol combining data hiding from the server with differentially-private aggregate computation at the server; they consider the different setting of n parties streaming data to a server computing an overall summation. While this is an excellent attempt at providing both types of privacy guarantees together, their model is inappropriate for our setting: they assume that a trusted party can run a setup phase for each specific set of clients that will contribute to an aggregate ahead of time. Besides the lack of such a trusted party in our model, importantly, one does not know during system setup which specific clients will pass through each sample point in the future; even the number n of such clients is unknown.

9. IMPLEMENTATION AND EVALUATION

End-to-end metric	Result
Setup	0.16 s
Join, Nexus client	0.92 s
Join, laptop client	0.42 s
Upload without account., Nexus	0.29 s
Upload with account., Nexus	2.0 s
Upload with 20% account., Nexus	0.6 s
Upload without account., laptop	0.094 s
Upload with account., laptop	0.84 s
Aggregation (10^3 samples)	0.2 s
Aggregation (10^4 samples)	0.46 s
Aggregation (10^5 samples)	3.1 s

Table 2: End-to-end latency measurements.

In this section, we demonstrate that PrivStats can run on top of commodity smartphones and hardware at reasonable costs. We implemented an end-to-end system; the clients are smartphones (Nexus One) or commodity laptops (for some social crowd-sourcing applications), the server is a commodity server, and the SM was evaluated on smartphones because it runs on clients. The system implements our protocols (Fig. 2) with SLP and enforcing a quota number of uploads at each aggregate per client for accountability.

We wrote the code in both C++ and Java. For our evaluation below, the server runs C++ for efficiency, while clients and SM run Java. Android smartphones run Java because Android does not fully support C++. (As of the time of the evaluation for this paper, Android NDK lacks support for basic libraries we require.) We implemented our cryptographic protocols using NTL for C++ and BigInteger for Java. Our implementation is ≈ 1300 lines of code for all three parties, not counting libraries; accountability forms 55% of the code. The core code of the SM is only 62 lines of code (not including libraries), making it easy to secure.

9.1 Performance Evaluation

In our experiments, we used Google Nexus One smartphones (1GHz Scorpion CPU running Android 2.2.2., 512 MB RAM), a commodity laptop (2.13 GHz Intel Pentium CPU 2-core, 3 GB RAM), a commodity server (2.53GHz Intel i5 CPU 2-core, 4 GB RAM), and a server with many cores for our scalability measurements: Intel Xeon CPU 16 cores, 1.60 GHz, 8 GB RAM. In what follows, we report results with accountability, without accountability (to show the overhead of the aggregation protocols), and with 20% of uploads using accountability as recommended in §6.3.

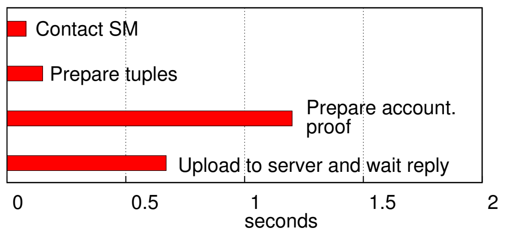


Figure 4: Breakdown of Nexus upload latency.

Server 0.29 s	Client laptop 0.46 s	Client Nexus 1.16 s

Table 3: Runtime of the accountability protocol.

Table 2 shows end-to-end measurements for the four main operations in our system (Fig. 2); these include our operations and network times. We can see that the latency of setup and join are insignificant, especially since they only happen once for the service or once for each client, respectively.

The latency of upload, the most frequent operation, is measured from the time the client wants to generate an upload until the upload is acknowledged at the server, including interaction with the SM. The latency of upload is reasonable: 0.6 s and up to 2 s for Nexus. Since this occurs either in the background or after the client triggered the upload, the user does *not* need to wait for completion. Figure 4 shows the breakdown into the various operations of an upload. We can see that the accountability protocol (at client and server) takes most of the computation time (86%). The cost of accountability is summarized in Table 3.

For aggregation, we used the Paillier encryption scheme which takes 33 ms to encrypt, 16.5 ms to decrypt, and 0.03 ms for one homomorphic aggregation on the client laptop, and 135 ms to encrypt and 69 ms to decrypt on Nexus. The aggregation time includes server computation and interaction with the SM. The latency of this operation is small: 0.46 s for 10^4 tuples per aggregate, more tuples than in the common case. Moreover, the server can aggregate samples as it receives them, rather than waiting until the end.

In order to understand how much capacity one needs for an application, it is important to determine the throughput and latency at the server as well as if the throughput scales with the number of cores. We issued many simultaneous uploads to the server to measure these metrics, summarized in Table 4. We can see that the server only needs to perform 0.3 s of computation to verify a cryptographic proof and one commodity core can process 860 uploads per minute, a reasonable number. We parallelized the server using an adjustable number of threads: each thread processes a different set of aggregate identifiers. Moreover, no synchronization between these threads was needed because each aggregate is independent. We ran the experiment on a 16-core machine: Fig. 5 shows that the throughput indeed scales linearly in the number of requests.

We proceed to estimate the number of cores needed in an application. In a social crowd-sourcing application, suppose that a client uploads samples on the order of around 10 times a day when it visits a place of interest (e.g., restaurant). In this case, one commodity core can already serve about 120,000 clients.

In the vehicular case, clients upload more frequently. If n is the number of cars passing through a sample point, a server core working 24 h can thus support $\approx 24 \cdot 60 \cdot 860/n$ statistics in one day. For instance, California Department of Transportation statistics [7] indicate that there are about $2,000$ cars on average passing in an hour through a highway lane. In this setting, one core supports

Server metric	Measurement
Upload latency, with account.	0.3 s
Upload latency, no account.	0.02 s
Throughput with 0% account.	2400 uploads/core/min
Throughput with 20% account.	860 uploads/core/min
Throughput with 100% account.	170 uploads/core/min

Table 4: Server evaluation for uploads. Latency indicates the time it takes for the server to process an upload from the moment the request reaches the server until it is completed. Throughput indicates the number of uploads per minute the server can handle.

Party	Join	Upload	Aggregation
Client	1 KB	7.01 KB	0
Server	0.5 KB	0.5 KB	128 B
SM	0	2 B	4B
Total	1.5 KB	7.5 KB	132 B

Table 5: The amount of data sent over the network.

about $\approx$ 620 different aggregates. Of course, the precise number depends on the application, but this estimation suggests our protocol is feasibly efficient.

We experimented with the SM as well; the throughput of s_{id} requests was mostly limited by how many http requests the smartphone can process. This is because the SM has very little work to do. The SM spends $\approx$ 140 ms for decryption and proof of decryption per aggregate (once per aggregate), and $<$ 5ms per s_{id} request on a smartphone (once per client), while the server performs 300 ms worth of checks on a commodity server (once per client), resulting in more than 50 times more work than the SM especially when considering the different device capacities.

10. BANDWIDTH AND STORAGE USAGE

We measure the amount of data sent by each party during our protocol; Table 5 shows that the message sizes are reasonable.

The size of a tuple consists of 6.875 KB for the accountability proof and 137 bytes for the other fields.

The client should not maintain state unless she wants to preprocess some tokens. The server should maintain all the accountability tokens (1024 bits each) received for an aggregate to make sure that the tokens uploaded are unique, a requirement for the accountability protocol. If the server performs the probabilistic accountability check (§6.3), it should store in disk accountability proofs for an aggregate that it did not check in case it discovers attempts to fraud for one of the aggregates. After an aggregation was computed, the server discards tokens and proofs. A 4 GB RAM can hold $32 \cdot 10^6$ such values. Each proof is 6.875KB, so 1 TB of disk storage could store $156 \cdot 10^6$ such proofs. The SM needs to maintain, for each aggregate, about 20 bytes for an aggregate id and a the number of tuples uploaded so far, and for all aggregates, a public key pair $<$ 600bytes. Thus, storage should not be a problem.

10.1 Accuracy and Effectiveness

In this section we evaluate the accuracy of our protocol and its robustness against malicious uploads. In the process, we justify recommended values for quota and U_{id}, the total number of tuples to be uploaded for aggregate id.

As mentioned in §5, we suggest $\mathsf{quota} = 3$. Our reasoning is the following: On the one hand, we want the quota to be as small as possible to limit the error malicious clients can introduce into the aggregate. On the other hand, a quota of 1 or 2 would make it difficult for clients to upload U_{id} tuples in total: for aggregates

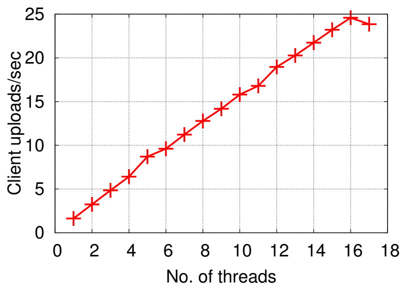


Figure 5: Throughput at the server versus number of parallel threads, showing linear scaling on a 16-core machine.

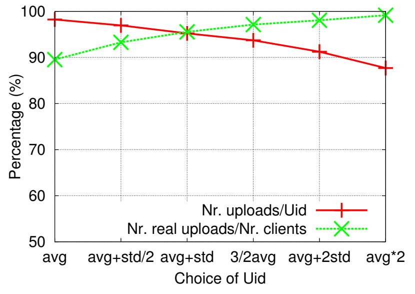


Figure 6: Total uploads at the server divided by U_{id}, and the number of clients that upload over total clients passing by a sample point.

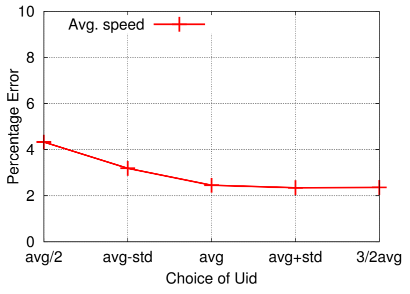


Figure 7: CarTel data showing the error in computing the average delay or speed for various choices of U_{id}.

with an unexpectedly low client turnout, participating clients need to upload more tuples in the place of missing clients to reach U_{id}.

We first analyze how much a single client can affect the aggregates by malicious uploads. Let N be the number of clients uploading for a certain aggregate, I the interval of accepted values, and μ the aggregate result when all clients are honest. For example, using the statistics in [7], N can be 2000 clients in an hour. For average speed computation, $I = (0, 100)$ mph and μ might be 60 mph, for instance. The highest change in the average that a client can induce is $\pm|I|\mathsf{quota}/N$ and the maximum fractional error is $\pm|I|\mathsf{quota}/N\mu$, where $|I|$ is the length of the interval I. We can see that if N is large (a popular area), the error introduced by a client is small. For our example, this is equal to ± 0.15 mph and 0.25% for $\mathsf{quota} = 3$, both rather small.

Next, we evaluate the accuracy of our scheme both analytically and empirically against real traces. We obtained real traces from the CarTel project testbed, containing an average of $\approx$ 400 one-day paths of taxi drivers in the Boston/MA area for each month of year 2008 driving mainly through the Boston area, but extending to MA, NH, RI, and CT areas. In-car devices report segment id (a segment is a part of a road between two intersections), time, average speed on the segment and average delay on the segment. We considered that an aggregate is computed on each segment in every hour. We restricted our attention to aggregates with at least 50 drivers, which is most often the case in practice. We averaged all the results reported below over each month of the year and over all aggregates considered.

As discussed in §7, for "count" aggregates we do not need the SM and we have no U_{id}. For non-count aggregates, the choice of U_{id} imposes a mild tradeoff between accuracy of aggregate result and privacy. A low U_{id} may not allow some drivers to upload (because at most U_{id} tuples must reach the server), while a large U_{id} may be hard for clients to achieve in cases with an unexpectedly low client turnout because each client can at most upload quota tuples. If the server receives a lower number of tuples than U_{id}, the server learns some information about the clients because the number of tuples uploaded is no longer independent of the number of tuples generated. If the number of tuples uploaded tends to be close to U_{id} for most cases, then little information is leaked.

U_{id} should be chosen as a function of the historical number of uploads at an aggregate point, as follows. In Figure 6, we vary U_{id} by using combinations of avg – the average number of uploads at a sample point in all our trace over a year – and std – the corresponding standard deviation. In practice, avg and std can be obtained based on historical data or estimates of the traffic at a sample point. The decreasing line represents to what extent clients can upload U_{id} in total; as expected, this decreases as U_{id} increases. The increasing line shows the percentage of clients passing through the sample point that can upload at least one tuple. Fortunately, we can see that for most values of U_{id}, both metrics are high, typically greater than 90%. This means that our U_{id}-upload protocol is feasible to achieve in practice and does not significantly affect the accuracy of the aggregate.

Examining Fig. 6, $U_{\mathsf{id}} = \mathsf{avg} + \mathsf{std}$ seems to be the best choice. We can gain some intuition as follows. Even if the turnout at a sample point is larger than avg by a standard deviation, most clients should still be able to upload. At the same time, U_{id} cannot be too much larger than avg for cases when there is a small turnout. We tried larger quotas of 4 and 5 and found that $U_{\mathsf{id}} = \mathsf{avg} + (\mathsf{quota} - 1)/2 \cdot \mathsf{std}$ still represents the intersection of the lines in Fig. 6.

Finally, we examine the accuracy error introduced by our protocol for this choice of U_{id}. Count aggregates of course introduce no error. For summation-type aggregates (see §7), the increasing line in Fig. 6 provides this answer. If every value summed comes from the same distribution, the error is roughly 5% which we consider reasonable, especially since summation-type averages are less common.

For average-type aggregation (§7), the accuracy is very high: even if only a subset of the clients may be able to upload in certain cases, this subset is chosen at random from all clients passing through the sample point due to the random timing in the sync. interval (See §5 and Fig. 3). Assuming each client's value is taken from the same distribution, the expected average of tuples is equal to the real expected average; moreover, due to the law of large numbers, for popular aggregates (which therefore also have larger U_{id}), the observed average will become very close to the real one. For aggregates with as few as ≈ 100 uploads per aggregate (as is the case in Cartel), the errors were already small: Fig. 7 shows that the error in average delay is at most 5% and the error in average speed is 3%.

We do not claim that CarTel traffic patterns are representative, but one can use a similar analysis of historical data to deduce proper U_{id} for other cases. In fact, we expect our protocol to be even more effective in practice because the CarTel network is very small (≈ 30 drivers); in a system with more participants, both metrics in Fig. 6 will simultaneously be larger, many values of U_{id} will be suitable, and errors of averages smaller due to the law of large numbers.

11. RELATED WORK

Aggregate statistics for location privacy. Systems such as [17], CliqueCloak [15], [20], [25], [21], and Triplines [19], address the same problem as PrivStats. Their approaches are to use trusted parties to aggregate time-consecutive driver uploads for the same location, to avoid uploading in sensitive areas, to distort spatially and temporally locational services, or to "cloak" the data using spatial and temporal subsampling techniques.

One problem with these solutions is that they neither provide rigorous privacy guarantees nor protect against side information. As discussed in §3, even if a server only sees anonymized tuples, considerable private information can leak due to SI. For example, in [21, 19], a trusted party aggregates the upload of each driver on a piece of road with the uploads of other $k-1$ drivers close in time and location to provide k-anonymity. While one individual tuple has k-anonymity, when intersecting the sets of k-anonymity for a path, the driver may not be necessarily k-anonymous over her entire path among other drivers and their paths. And of course SI can cause further violations of k-anonymity.

Another difficulty with most of these solutions is their reliance on fully trusted intermediaries between drivers and server. For example, trusted parties in [17, 15, 19, 20] receive tuples with clients identifiers in plaintext and are supposed to remove them from the tuple. If these parties are compromised or collude with other trusted parties in the system, so are the paths of the drivers, as they have access to driver identifiers. In our work, if the SM gets compromised, the server will still only see anonymized tuples (see §8), and the use of the SM achieves stronger security.

Moreover, accountability has previously either been ignored (allowing drivers to bias significantly the aggregate result) [17, 15, 25, 21] or handled by having tuples contain driver identifiers [19, 20]. For instance, in [19, 20], users upload tuples containing their identities to a trusted party that checks if clients have uploaded too much, while another trusted party performs aggregation. If these trusted parties collude, driver paths with identities leak.

Other related work. We now discuss systems for protecting location privacy that target different but related problems.

VPriv [34] and PrETP [2] compute functions over a particular driver's path as opposed to computing statistics over the paths of *multiple* clients; for instance, they can compute the total toll that driver Alice has to pay based on how much she drove in a month. They associate the result of a function to a client's id, whereas PrivStats keeps clients anonymous and computes statistics over all clients. VPriv and PrETP also use the general zero-knowledge concept, but the actual cryptographic proofs and protocols are different from ours and inapplicable to our setting: in VPriv and PrETP, all the clients who provide data for a function must be online and perform work at the same time, which is impractical for aggregates with hundreds of clients. Moreover, these protocols do not support aggregates over data from different clients.

Furthermore, VPriv is vulnerable to side information because the server sees anonymized tuples with time and location information. VPriv acknowledges this problem, but does not solve it, leaving it for future work. PrETP also suffers from SI, but to a lesser extent than VPriv. Finally, VPriv and PrETP do not provide our type of accountability: clients can upload as many times as they desire. They have a random spot check scheme involving a trusted party that makes sure that clients upload at least once per location. Our accountability protocol uses no trusted parties, leaks no privacy, and most importantly performs different enforcements.

SEPIA [6] uses special-purpose secure-multiparty protocols to compute aggregate statistics over network traffic. However, the approach in SEPIA is not plausible in our usage scenario. Some trusted "input peers" provide network data, and a cluster of "privacy peers" performs all the computation on this data. The protocol relies on a majority of the privacy peers to not collude. In contrast to our system, these peers perform almost all computation, and if they collude, all client data is visible. Furthermore, input peers are trusted, so SEPIA does not provide any accountability checks against abusive clients. SEPIA also does not hide the number of tuples for each aggregate.

There has also been recent work by Narayanan et al. [31], Zhong et al. [44], and Puttaswamy and Zhao [36] on preserving location privacy in location-based social networking applications, an area also full of location privacy concerns. This work has a different goal than ours: it allows users to exchange location-based information with friends while protecting their privacy (against disclosure to both the server and to friends).

Shokri et al. [41] provide a framework for quantifying location privacy as a way of comparing various location privacy schemes. Given a certain adversarial goal and models of adversarial knowledge and of a location privacy scheme, the framework enables quantification of the performance of the adversary. The authors also argue that side information (which they call "prior knowledge") should be considered when analyzing privacy. However, this work has a different goal than PrivStats, does not provide a definition of privacy resilient to side information, and does not propose protocols for accountable and location private computation of aggregate statistics.

Work on e-cash [9] is related to our accountability protocol (§6): one might envision giving each client a number of e-coins to spend on every aggregate. This approach fits well in spam control applications [1, 42]. However, it is not practical in our setting: coins must be tied to a particular location, time interval, and sample type, which requires generating a prohibitively large number of coins. Other work [8] gives each user n coins for each time period. Again, we need the coins tied to sample points, so this is not feasible. Also, e-cash adds a lot of complexity (and thus slowdown) to identify double-spenders, which we do not require. Our accountability protocol is simple, specific to our setting, and fast.

Finally, we remark that our approach complements work on differential privacy [12], as we discussed in §8.

12. CONCLUSION

In this paper, we presented PrivStats, a system for computing aggregate statistics for mobile, location-based applications that achieves both strong guarantees of location privacy and protection against cheating clients. PrivStats solves two major problems not solved by previous work: it ensures that no further information leaks even in the face of arbitrary side information attacks, and it provides client accountability without a trusted party. We implemented PrivStats on commodity phones and servers, and demonstrated its practicality.

13. ACKNOWLEDGMENTS

We thank Nickolai Zeldovich, Mike Walfish, Arvind Narayanan, and the anonymous reviewers for useful comments and feedback. This work was supported in part by NSF grants 0931550 and 0716273.

14. REFERENCES

[1] M. Abadi, A. Birrell, M. Burrows, F. Dabek, and T. Wobber. Bankable postage for network services. In *ASIAN*, 2003.
[2] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. PrETP: Privacy-preserving electronic toll pricing. *Usenix Security*, 2010.
[3] M. Bellare and O. Goldreich. On defining proofs of knowledge. *CRYPTO*, 1992.
[4] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, 2009.
[5] F. Boudot. Efficient proofs that a committed number lies in an interval. *EUROCRYPT*, 2000.

[6] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. *Usenix Security*, 2010.
[7] California Department of Transportation. Caltrans guide for the preparation of traffic impact studies.
[8] J. Camenisch, S. Hohenberger, M. Kohlweiss, A. Lysyanskaya, and M. Meyerovich. How to win the clonewars: Efficient periodic n-times anonymous authentication. In *CCS*, 2006.
[9] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Balancing accountability and privacy using e-cash. *Security and Cryptography for Networks*, 2006.
[10] J. Camenisch and A. Lysyanskaya. A Signature Scheme with Efficient Protocols. *Security and Cryptography for Networks*, 2002.
[11] I. Damgard and E. Fijisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. *ASIACRYPT*, 2002.
[12] C. Dwork. Differential privacy: A survey of results. In *TAMC 1-19*, 2008.
[13] E-ZPass. How it works. http://www.ezpass.com/index.html.
[14] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. *CRYPTO*, 1986.
[15] B. Gedik and L. Liu. Location privacy in mobile systems: A personalized anonymization model. In *ICDCS*, 2005.
[16] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. *Symposium on the Theory of Computation*, 1985.
[17] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *MobiSys*, 2003.
[18] M. Gruteser and B. Hoh. On the anonymity of periodic location samples. In *IEEE Pervasive Computing*, 2005.
[19] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. Bayen, M. Annavaram, and Q. Jacobson. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *Mobisys*, 2008.
[20] B. Hoh, M. Gruteser, H. Xiong, and A. Alrabady. Enhancing security and privacy in traffic-monitoring systems. In *IEEE Pervasive Computing*, 2006.
[21] B. Hoh, M. Gruteser, H. Xiong, and A. Alrabady. Preserving privacy in GPS traces via uncertainty-aware path cloaking. In *CCS*, 2007.
[22] B. Hull, V. Bychkovsky, K. Chen, M. Goraczko, A. Miu, E. Shih, Y. Zhang, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. *Sensys*, 2006.
[23] N. Husted and S. Myers. Mobile location tracking in metro areas: Malnets and others. In *CCS*, 2010.
[24] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, 1997.
[25] J. Krumm. Inference attacks on location tracks. In *IEEE Pervasive Computing*, 2007.
[26] J. Lowensohn. Apple sued over location tracking in iOS. http://news.cnet.com/8301-27076_3-20057245-248.html, 2011. CNET News.
[27] U. Maurer. Unifying Zero-Knowledge Proofs of Knowledge. *AFRICACRYPT*, 2009.
[28] E. Mills. Google sued over Android data location collection. http://news.cnet.com/8301-27080_3-20058493-245.html, 2011. CNET News.
[29] Mobile Millennium. http://traffic.berkeley.edu/.
[30] M. Mun, S. Reddy, K. Shilton, N. Yau, P. Boda, J. Burke, D. Estrin, M. Hansen, E. Howard, and R. West. PEIR, the personal environmental impact report, as a platform for participatory sensing systems research. In *MobiSys*, 2009.
[31] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. *NDSS*, 2011.
[32] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.
[33] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. *CRYPTO*, 1991.
[34] R. A. Popa, H. Balakrishnan, and A. J. Blumberg. VPriv: Protecting privacy in location-based vehicular services. *Usenix Security*, 2009.
[35] G. Poupard and J. Stern. Security analysis of a practical "on the fly" authentication and signature generation. *EUROCRYPT*, 1998.
[36] K. Puttaswamy and B. Zhao. Preserving privacy in location-based mobile social applications. *International Workshop on Mobile Computing and Applications*, 2010.
[37] R. Reid. TomTom admits to sending your routes and speed information to the police, 2011. CNET UK.
[38] P. Riley. The tolls of privacy: An underestimated roadblock for electronic toll collection usage. In *Third International Conference on Legal, Security, and Privacy Issues in IT*, 2008.
[39] C. P. Schnorr. Efficient identification and signatures for smart cards. *CRYPTO*, 1989.
[40] E. Shi, T.-H. H. Chan, E. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *NDSS*, 2011.
[41] R. Shokri, G. Theodorakopoulos, J.-Y. L. Boudec, and J.-P. Hubaux. Quantifying location privacy. In *IEEE Symposium on Security and Privacy*, 2011.
[42] M. Walfish, J. Zamfirescu, H. Balakrishnan, D. Karger, and S. Shenker. Distributed quota enforcement for spam control. In *NSDI*, 2006.
[43] WMUR. Police: Thieves robbed home based on Facebook, 2010. http://www.wmur.com/r/24943582/detail.html.
[44] G. Zhong, I. Goldberg, and U. Hengartner. Louis, Lester, and Pierre: Three protocols for location privacy. International Conference on Privacy-Enhancing Technologies, 2007.

APPENDIX

A. THE PROBLEM OF SYNCHRONIZATION

In this section, we show how hiding the number of tuples could be reduced, under certain model assumptions, to a distributed algorithms problem that can be shown impossible. Of course, this is not a proof that the problem is impossible in our setting, because one could always change the model or weaken some privacy definition, but it provides strong intuition that this might be so practically.

In our setting, a set of clients pass by a sample point: these clients do not know of each other, and, in particular, they do not know how many they are. With high probability, we would like each client to send at least one tuple but no more than a constant quota to the server, and we would not like the server to know how many clients there are. The malicious server can also behave as a client passing through the sample point. There can be many clients in the system and it is infeasible to try to contact them all. Considering there is no trusted party, then any party they use is untrusted, so any packets clients send out can be considered to pass through the server. One insight into the reduction is that the clients must be running the same "upload" algorithm, albeit with different inputs (e.g., their identifiers), the clients thus being modeled as uniform random processes.

Distributed algorithms problem: A network contains a set of n uniform random processes, where n is arbitrary and unknown, and an untrusted server. Each process is connected to the server with an anonymous link. Executions proceed in synchronous rounds. In each round, each process chooses whether or not to send a message to the untrusted server using its anonymous link. The process must guarantee w.h.p. that every process sends at least one message to the server but no more than quota. Their goal is to satisfy this condition while preventing the server from learning n.

Formally, let $p_{i,j}$ be the probability that the server guesses the network is of size i given a network of size j. For every i, j, k: $p_{i,j} = p_{i,k}$. (Other looser definitions of knowledge work as well: our below theorem is strong.)

CLAIM 4. *For every random process definition that guarantees w.h.p. that every process sends at least one message and at most* quota *to the server, we can construct a server that can guess the size of the network within a constant factor, w.h.p.*

PROOF. Given the definition of the random process the adversary can identify the first round in which it has some non-zero probability of sending a message. Call this round r and the probability p. Because the processes are uniform, this probability is the same for all processes in the network. Let c be the number of messages received by the server in round r in a given execution with network size n. Using the Chernoff bound, the guess that there are c/p clients will be within a constant factor of n, w.h.p. □