Towards Statistical Queries over Distributed Private User Data

Ruichuan Chen[†] Alexey Reznichenko[†] Paul Francis[†] Johannes Gehrke[§]
[†]*Max Planck Institute for Software Systems (MPI-SWS), Germany*
[§]*Cornell University, Ithaca, NY 14853, USA*
{*rchen, areznich, francis*}*@mpi-sws.org, johannes@cs.cornell.edu*

Abstract

To maintain the privacy of individual users' personal data, a growing number of researchers propose storing user data in client computers or personal data stores in the cloud, and allowing users to tightly control the release of that data. While this allows specific applications to use certain approved user data, it precludes broad statistical analysis of user data. Distributed differential privacy is one approach to enabling this analysis, but previous proposals are not practical in that they scale poorly, or that they require trusted clients. This paper proposes a design that overcomes these limitations. It places tight bounds on the extent to which malicious clients can distort answers, scales well, and tolerates churn among clients. This paper presents a detailed design and analysis, and gives performance results of a complete implementation based on the deployment of over 600 clients.

1 Introduction

User privacy on the Internet has become a major concern. User data is exposed to organizations in a bewildering and growing variety of ways. Sometimes the exposure is known to users, as when a user makes a purchase in an online store, or updates a profile on an online social networking site. But often users are unaware that their data is being exposed, for instance by third party trackers [34] or smart phone applications [19]. In some cases, the exposure provides benefits to users, for instance in the form of personalized recommendations. In other cases, the exposure may be detrimental to users, for instance as when Google used users' Gmail contact data to seed its Buzz social network [2].

There is a growing sense that this loss of privacy has to be brought under control. ProjectVRM, for instance, states that users must have exclusive control of their own data, and must be able to share data selectively or voluntarily [4]. This "user-owned and operated" principle has already been reflected in commercial and research efforts on various types of user data, such as individuals' healthcare data [3], time-series data [45, 49], and behavioral data used for advertising [7, 31, 51]. Here, each individual user's data is stored in a client or cloud device under the user's control, and is released in a controlled, limited, or noisy fashion. It is important to be able to make statistical queries over such *distributed* user data while still preserving privacy.

One general approach to supporting privacy-preserving statistical queries is to anonymize the user data or add noise to the user data, so that individual users cannot be identified. Unfortunately, this approach heavily restricts the utility of user data [43], and often users can be de-anonymized [1, 11, 41, 46], for instance using auxiliary information.

Another general approach, which we adopt in this paper, is to add noise to the answers of queries, in such a way that the privacy of individual users is protected. An instance of this approach that is popular in the research community is *differential privacy* [13, 14, 17]. Specifically, differential privacy adds noise to the answers of queries to statistical databases so that the querying system cannot detect the presence or absence of a single user or a set of users. Differential privacy provides a provable foundation for measuring privacy loss regardless of what information an adversary may possess. In spite of the fact that an increasing number of queries can lead to increased privacy loss, we find differential privacy to be an attractive model both because it does provide measurable privacy, and because there is substantial ongoing effort in developing its uses [32, 38, 39, 40, 47] and understanding its limitations [15].

Most work in differential privacy assumes a centralized database front-ended by a trusted query module. There is, however, no centralized database existing in a distributed setting with individual users maintaining their own data. Some form of *distributed differential privacy* is therefore required. To our knowledge, there are

a few prior designs for distributed differential privacy in the literature [16, 30, 45, 49], none of which appear practical in a realistic distributed environment. The first design [16] has a per-user computational load of $O(U)$, where U is the number of users, making it impractical for large systems. Following designs [45, 49] reduce the per-user computational load from $O(U)$ to $O(1)$, but this complexity assumes that key shares have been distributed among users, using an expensive secret sharing protocol. However, in reality users often exhibit churn (go on and offline), and it would be very hard to execute such a protocol among a sizable population of such users. To tolerate churn, the recent design [30] introduces two honest-but-curious servers to collaboratively compute the query result. Nevertheless, these designs [30, 45, 49] all suffer from a common attack that even a single malicious user can substantially distort the query result.

The goal of this paper is to design, build, and measure a *practical* system that provides differentially private query semantics over distributed user data. Our system, dubbed **PDDP** for Practical Distributed Differential Privacy, assumes that *clients* are user devices under users' control. Therefore, they may be malicious, and they are not always online. An *analyst*, who wishes to make statistical queries over some number of clients, formulates a query and transmits it to an honest-but-curious *proxy*[1], which further forwards it to the specified number of clients. Each client locally executes the query, and sends its answer back to the proxy encrypted with the analyst's public key. In parallel, the proxy and clients, using an efficient XOR homomorphic bit-cryptosystem, *collaboratively* generate a set of additional noisy answers that produce the required amount of differentially private noise. The proxy then shuffles the received client answers and the indistinguishable noisy answers, and forwards them together to the analyst. Finally, the analyst decrypts them and computes the statistical result under the differentially private guarantee. Altogether, this paper makes the following contributions:

- It proposes what is to our knowledge the first distributed differentially private system, PDDP, that is practical in that it can operate at large scale with malicious clients and under churn.
- It gives a detailed privacy analysis, and presents the implementation results of a fully functional PDDP system with a realistic deployment of 600+ clients.

The rest of this paper is organized as follows. We first give the security assumptions and performance goals in §2. The system design is presented in §3. We then provide a privacy analysis of the system in §4. This is followed in §5 by the implementation description and performance evaluation based on micro-benchmarks and our 600+ client deployment. In §6, we sketch a design for how to weaken the proxy trust requirement by using trusted hardware at proxy. Finally, we give an overview of related work in §7, and outline future work in §8.

[1] We later suggest a practical approach to reduce the proxy trust (§6).

2 Assumptions and Goals

The PDDP system consists of three components: *analysts*, *clients*, and *proxy*. Analysts make queries to the system, and collect answers. Clients locally maintain their own data, and answer queries. The proxy mediates between the analysts and clients, and adds differentially private noise to clients' answers to preserve privacy.

2.1 Security Assumptions

Analysts are assumed to be potentially malicious, with a goal of violating individual users' privacy. An analyst may collude with other analysts, or pretend to be multiple distinct analysts. An analyst may take control of clients, and attempt to use the PDDP protocol to reveal information about those clients. (Of course, an analyst could reveal information about those clients outside of PDDP, but this is the case today with for instance malware and so is out of scope.) An analyst may deploy its own clients and manipulate their answers. An analyst may also publish its collected answers. Analysts can intercept and modify all messages (e.g., an ISP posing as an analyst).

Clients are also assumed to be potentially malicious, with a goal of distorting the statistical results learned by analysts. Clients may generate false or illegitimate answers under coordinated control (e.g., as a botnet), and may act as Sybils [12].

The proxy is assumed to be honest but curious (HbC). It will faithfully follow the specified protocol, but may try to exploit additional information that can be learned in so doing. The proxy does not collude with other components. We discuss how we may be able to relax the HbC assumption by using trusted hardware in §6.

It is assumed that clients have the correct public keys for analysts and the proxy, that analysts and the proxy have correct public keys for each other, and that the corresponding private keys are kept secure.

Discussion. It would be ideal if our design did not require an HbC proxy. Indeed, some prior designs for distributed differential privacy do not require such a proxy [16, 45, 49]. They achieve this, however, at an unacceptable cost in a realistic distributed setting (see §1). Here we briefly discuss the viability of an HbC proxy.

We envision a scenario whereby the analysts pay the proxy to operate, as suggested in [24]. While this admittedly leads to opportunities for collusion, we point out

that such relationships already exist in industry today that normally do not result in collusion. For instance, companies pay for independent audits of their financial books or independent safety testing of their products, even though this may lead to negative consequences. We therefore believe that, in practice, the HbC arrangement is reasonable for the PDDP system in most scenarios.

2.2 Client Characteristics

The client population consists of user devices, including home and mobile devices. Clients are therefore assumed to have the churn characteristics, and the CPU and bandwidth capacities of current home and mobile devices. In other words, clients may go offline or shutdown at any time, and may have limited resources for computation and data transmission.

2.3 Goals

The primary goal of our PDDP system is that the differentially private guarantee is *always* maintained for every honest client.

The second goal is that the *maximum* absolute distortion in the final statistical result tallied by an analyst is bounded by the number of malicious clients (here ignoring distortion due to the differentially private noise itself). In other words, if z clients are malicious, then the absolute error in the statistical result will be z or less.

Finally, the system should scale for queries to a very large client base (millions of clients). While client churn may result in it taking a relatively long time to produce statistical results, this should not prevent results from being produced.

3 System Design

3.1 Differential Privacy Background

In principle, differential privacy ensures that the output of a computation does not violate the privacy of individual inputs. Formally, a computation $\mathscr{F}$ gives (ε, δ)-differential privacy [16] if, for all datasets D_1 and D_2 differing on one record, and for all outputs $S \subseteq Range(\mathscr{F})$:

$$\Pr[\mathscr{F}(D_1) \in S] \leq \exp(\varepsilon) \times \Pr[\mathscr{F}(D_2) \in S] + \delta \qquad (1)$$

In other words, for any possible record, the probability that the computation $\mathscr{F}$ produces a given output is almost independent of whether or not this record is present in the dataset.

The strong guarantees of differential privacy do not come for free. Privacy is preserved by adding noise to the output of a computation. Specifically, there are two privacy parameters: ε and δ. The former parameter ε mainly controls the tradeoff between the accuracy of a computation and the strength of its privacy guarantee. Higher ε represents higher accuracy but weaker privacy guarantee, and vice versa. The latter parameter δ relaxes the strict relative shift of probability in some cases, where the expression (1) cannot be satisfied without a non-zero δ [16].

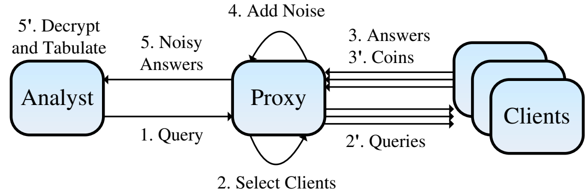

Figure 1: System components and basic protocol

Differential privacy does not make any assumptions about the adversary. It is independent of the adversary's computational power and auxiliary information. Such information has been shown to break many other privacy mechanisms [11, 41, 46].

3.2 Basic Protocol Design

3.2.1 Periodic Client-Proxy Exchange

Periodically, each online client connects to the proxy using standard session encryption (e.g., TLS) authenticated by the proxy's public key. On the first such connection, the proxy assigns a unique ID to the client, which the (honest) client uses to identify itself in subsequent connections. With each connection, the client receives and answers any pending queries from the proxy. In addition, the client sends zero or more random "tossed coins" (i.e., encrypted '0' or '1' values) to the proxy (see §3.2.3).

The proxy maintains a complete list of clients. For each client, it stores the client's ID, the client's *privacy deficit* (an indication of the client's privacy loss across all analysts, see §3.3.2), and the timestamp of when the client last connected. Clients that have not connected for a long time (weeks or months) are removed from the list.

3.2.2 Query-Answer Workflow

The query-answer workflow consists of five steps, as illustrated in Figure 1.

Step 1: Query Initialization (Analyst→Proxy). An analyst formulates a general SQL-style *query*, and transmits it to the proxy. Answers to a query will be provided as a set of b *buckets* whose lower bounds L_i and upper bounds U_i are specified by the analyst. Different buckets should not overlap. Additionally, the analyst also specifies the number of clients c that should be queried, and a privacy parameter ε:

$$A \rightarrow P : query, \{L_i, U_i\}_{i=1}^{b}, c, \varepsilon$$

Here we again assume standard session encryption to mutually authenticate proxy and analyst, and prevent man-in-the-middle attacks from distorting messages. As an illustrative example, the analyst's query "what is the distribution of ages among males?" can be formulated as an SQL-style query "`SELECT age FROM info WHERE gender='m'`" with 4 buckets, e.g., age 0∼12, age 13∼20, age 21∼59, and age≥60.

Step 2: Query Forwarding (Proxy→Client). Once the proxy receives the analyst's query, it rejects the query if c is too low or too high, or if ε exceeds the maximum allowable privacy level. Otherwise, the proxy selects c unique clients to which to send the query, under one of the following two policies (selectable by the analyst):

- Select c clients randomly from the complete list of clients, and wait for them to connect.
- Select the first c clients that connect.

Under the first policy, some of the selected clients may not connect after a long time (many hours or days). In this case, other random clients can be selected instead, until answers from c or nearly c clients are collected. Under the second policy, answers can be collected more quickly, but with a bias towards clients that are more often online. The proxy may reject client connections that occur more frequently than the defined connection period. This is to prevent malicious clients from connecting very frequently and so dominating the answer set.

After client selection, the proxy forwards the query (with bucket information) to the c selected clients when they connect to the proxy:

$$P \rightarrow C : query, \{L_i, U_i\}_{i=1}^{b}$$

Step 3: Client Response (Client→Proxy). Each client maintains its own data in a local database. Upon receiving a query, the client produces an *answer* in the form of a '1' or a '0' per bucket, depending on whether or not the answer falls within the range of that bucket. Depending on the query, more than one bucket may contain a '1'.

Each per-bucket binary value is individually encrypted using the Goldwasser-Micali (GM) bit-cryptosystem [27, 28] with the analyst's public key. The client returns this series of encrypted binary values $\{v_i\}_{i=1}^{b}$ as the actual answer to the proxy:

$$C \rightarrow P : \{v_i\}_{i=1}^{b}$$

GM is a probabilistic public-key cryptosystem. A given input produces a different ciphertext every time it is encrypted. GM has significant advantages in our system. First, it is very efficient compared with other more general public-key cryptosystems. Second, it is a single-bit cryptosystem which encodes only two possible values, 0 and 1. This enforces a binary value in each bucket, and prevents the use of individual encrypted values as a covert channel.

Upon receiving a client's answer (in the form of a series of GM-encrypted values $\{v_i\}_{i=1}^{b}$), the proxy checks the legitimacy of the answer. A legitimate GM-encrypted value must have its Jacobi symbol equal to '+1', so that the proxy can easily and efficiently detect illegitimate values [18, 48]. If a client's answer is legitimate (i.e., $\{v_i\}_{i=1}^{b}$ are all legitimate), the proxy stores the answer locally; otherwise, the proxy discards this answer.

Step 4: Differentially Private Noise Addition. To preserve clients' privacy, the proxy adds differentially private noise to each bucket. This is done by creating some number of additional binary noise-values selected from a binomial distribution with success probability $p = 0.5$. We call these additional noise-values *coins*. Enough *unbiased* coins must be added by the proxy to obtain the amount of noise required by the differential privacy. It is proven in [16] that forming a binomial distribution by adding n unbiased coins achieves (ε, δ)-differential privacy, where

$$n \geq \frac{64 \ln(\frac{2}{\delta})}{\varepsilon^2} \tag{2}$$

The value of ε is chosen by the analyst according to some accuracy/privacy trade-off (see §3.1). The value of δ may also be chosen by the analyst, but if $\delta \geq 1/c$, a single client's privacy may be compromised [33]. This may happen in the case where the query is for an attribute unique to a single client, e.g., its social security number. Applying $\delta < 1/c$ to expression (2), the number of coins for each bucket is at least:

$$n = \lfloor \frac{64 \ln(2c)}{\varepsilon^2} \rfloor + 1 \tag{3}$$

The proxy maintains a *pool* of unbiased coins (see §3.2.3). Note that all coins in this pool are GM-encrypted with the analyst's public key, and are indistinguishable from clients' answer values.

If a query has b buckets, then the proxy needs to use (and remove) in total $b \times n$ coins from the pool, i.e., to add n different coins to each bucket.

Step 5: Noisy Answers to Analyst (Proxy→Analyst). With the noise addition, in each bucket, there are $c+n$ encrypted binary values, c from clients' answers and n from added coins. After a random delay, the proxy further shuffles the $c+n$ values in each bucket *independently*. This prevents a client from being identified based on the vector of 1's and 0's in its answer, and eliminates the potential covert channel. The result is a set of shuffled encrypted values $\{e_{i,j}\}_{j=1}^{c+n}$ for each i-th bucket.

Note that, in each bucket, adding n unbiased coins introduces differentially private noise with mean equal to $n/2$. As a result, for the final aggregate answer adjustment, the proxy also informs the analyst of n, i.e., the number of coins added to each bucket. Thus, the message sent from proxy to analyst is as follows:

$$P \rightarrow A: \left\{ \left\{ e_{i,j} \right\}_{j=1}^{c+n} \right\}_{i=1}^{b}, n$$

After the proxy sends this message to the analyst, the proxy adds ε and δ to the respective privacy deficits (see §3.3.2) of each client that contributed an answer. Here, δ can be approximately considered as $1/c$ (see Step 4).

Upon receiving the message, the analyst uses its private key to decrypt all encrypted binary values, and obtains a set of plaintext values $\{d_{i,j}\}_{j=1}^{c+n}$ for each i-th bucket. Finally, for each i-th bucket, the analyst sums up all plaintext values, and then subtracts $n/2$ to get the (noisy) aggregate answer a_i:

$$a_i = \sum_{j=1}^{c+n} d_{i,j} - \frac{n}{2} \tag{4}$$

In the end, the analyst obtains a noisy answer for how many clients fall within each bucket (i.e., a_i for the i-th bucket) under the guarantee of differential privacy. Ultimately, our design transforms any query into a counting query. Though this kind of query is simple, it can potentially be extended to support a large range of statistical learning algorithms [9, 10].

3.2.3 Coin Pool Generation

In the step of differentially private noise addition (Step 4), we assume that there is a pool of unbiased coins (per each individual analyst) maintained at the proxy. In this section, we describe how to generate this pool of coins.

The most straightforward way would be to let the proxy take full responsibility for generating the coins. However, this would allow the curious proxy to know the true noise-free aggregate answer should the analyst choose to publish its noisy aggregate answer (see §4.3). As an alternative, the clients could generate the required unbiased coins and send them to the proxy. However, in the absence of an expensive protocol such as secure multi-player computation [26, 52], malicious clients could generate biased coins.

In our system, the proxy and clients generate the unbiased coins collaboratively (see Figure 2). Each online client itself *periodically* generates an encrypted unbiased coin $\mathscr{E}(o_c)$ with an analyst's public key, and sends it to the proxy.

Of course, malicious clients could still generate biased coins. To defend against this, the proxy does two things. First, when the proxy receives a client-generated encrypted coin $\mathscr{E}(o_c)$, it verifies the legitimacy of the coin by checking its Jacobi symbol [18, 48] (as in Step 3). If this fails, the coin is dropped. Second, the proxy blindly *re-flips* the coin $\mathscr{E}(o_c)$ by multiplying it with the proxy's locally generated unbiased coin $\mathscr{E}(o_p)$, plus a modulo operation. This blind encrypted-coin re-flipping is possible because of the *XOR-homomorphic* feature of the GM cryptosystem we use: $\mathscr{E}(o_c) \times \mathscr{E}(o_p) \bmod m = \mathscr{E}(o_c \oplus o_p)$, where m is a part of the analyst's public key, and '$\oplus$' is the exclusive-or operator. This feature ensures that the modulus product $\mathscr{E}(o_c \oplus o_p)$ is an encrypted unbiased coin, regardless of any bias in the client's original coin. Note that the proxy does not know the actual (decrypted) value of the generated unbiased coin. Once generated, the proxy stores the unbiased coin in the locally maintained pool.

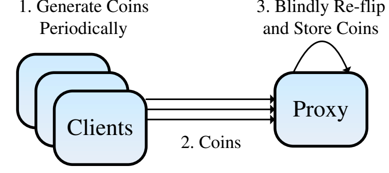

Figure 2: Unbiased Coin Generation

3.3 Practical Considerations

3.3.1 Utility of Aggregate Answer

The utility of the final aggregate answer learned by an analyst depends directly on the amount of added noise. The n coins added by the proxy (see expression 3), followed by the analyst's adjustment on the mean of $n/2$ (see expression 4), form a binomial distribution, which is a good approximation to the normal distribution $N(0, n/4)$.

The normal distribution is convenient in understanding the effect of added noise on the aggregate answer. In particular, the "68-95-99.7 rule" of the normal distribution tells us that the noisy aggregate answer falls within one, two, and three standard deviations away from the true noise-free answer with the probability of 68%, 95%, and 99.7%, respectively, where the standard deviation $\sigma = \sqrt{n}/2$. To give a concrete example, imagine that $c = 10^6$ clients, and a conservative privacy parameter of $\varepsilon = 1.0$ [39] is chosen. Given the normal distribution, for each bucket, there is a 68% probability that the noisy aggregate answer is within 15.24 away from the true answer, a 95% probability that it is within 30.48 away from the true answer, and a 99.7% probability that it is within 45.72. This gives high accuracy relative to the number of clients queried, and therefore the noisy aggregate answer has high utility even under a conservative privacy parameter setting.

3.3.2 Privacy Deficit vs. Privacy Budget

Under the definition of (ε, δ)-differential privacy, privacy loss is cumulative. This has given rise to the notion of a privacy budget [15, 17], where additional queries are simply not allowed after the cumulative ε and δ have reached some limit. This notion is completely impractical for our setting, where a user's personal data may persist for a lifetime or even longer. Whether or not to "kill" a database after some budget is reached is a policy decision. We prefer to treat the cumulative ε and δ as an ongoing measure of privacy loss rather than a hard limit (Step 5). As such, we refer to this measure as the *privacy deficit* rather than a privacy budget.

The assumption of cumulativity in differential privacy is very pessimistic because it effectively assumes that the correlation between answers to different queries is 100%, in other words, equivalent to repeating the same query (to the same statistical database or, in our case, to the same set of clients). While it could in theory be that every query is 100% correlated, in practice, many queries may not be very correlated. Furthermore, it is possible to informally estimate the amount of correlation between queries. For instance, the query "what percentage of users are male?" is highly correlated with the query "what percentage of users wear men's shoes?", but (probably) poorly correlated with the query "what percentage of users have spaghetti as their favorite food?".

Other factors also contribute to making queries not very correlated. In our setting, it is normally the case that only a fraction of clients actually answer a given query, and the set of clients changes from query to query. In addition, some of the data held at clients may change over time, making correlation between earlier and later queries less likely.

Given all the above, we can treat the privacy deficit as a *worst-case* measure of privacy loss. The actual privacy loss can be roughly estimated by taking into account the above factors.

Practical Approach. One way to leverage the privacy deficit would be to charge analysts accordingly, as suggested in [24]. In the case of PDDP, this charge would be a function of $c \times \varepsilon$, i.e., the total amount of increased privacy deficits across all queried clients. This would incentivize analysts to minimize the number of queries, the number of queried clients c, and the privacy parameter ε, while still obtaining high-utility answers. The best strategy for this depends on the application domain.

Note also that clients may locally maintain their own privacy deficits, and are free to not answer queries if some limit has been reached (or for any other reasons). Enabling this requires that the query sent from proxy to clients in Step 2 contain ε and δ (or c, if the value of δ is based on c).

3.3.3 Non-numeric Queries

Our SQL-style queries with associated buckets allow for a rich variety of sophisticated queries that produce numeric answers. Unfortunately, not all queries that an analyst may wish to make are numeric in nature. For instance, for the query "which website do you visit most often?", a client's answer may be one out of millions of websites. While non-numeric, this query can be transformed into a numeric query by mapping each website an analyst wishes to learn about into a numeric value. Unfortunately, this may produce a large number of buckets. This can often be partially mitigated in practice, for instance in this case by limiting the answer to the top 5000 most popular websites, as well as a "none of the above" bucket.

3.3.4 Sybils and NATs

The design, as described in §3.2, is susceptible to Sybil attacks [12], whereby a single client machine masquerades as multiple clients. In PDDP, the proxy can deal with this by limiting the number of clients selected at a single IP address for a given query. This may have the effect of biasing answers, for instance towards home users, where there are relatively few machines behind a NAT, and away from business users, where many users may be behind a NAT. This limit could be set by the analyst, thus giving it some control over this bias at the expense of a higher risk of answer skew by Sybil clients.

3.3.5 Scaling Considerations

To scale the PDDP proxy to millions of clients and queries, we propose a two-tier approach. The *front-end* devices at the proxy are responsible for individual clients. They exchange messages with clients and maintain the per-client states such as privacy deficits and last-connected timestamps. These devices scale easily through the addition of more front-end devices and the databases that support them. This is because there is no need for interaction between these devices other than the pairwise interaction needed for redundancy. An IP load-balancing router can be used to steer clients to the appropriate front-end devices.

The *back-end* devices at the proxy are responsible for individual queries. Each back-end device maintains a list of client IDs and the corresponding front-end devices that service these clients. This list does not need to be a complete list of all clients. It is enough if the list is a representative random selection of clients, but big enough to handle queries with a large target population. A query is submitted to a back-end device, which selects a set of clients, and passes the query to the appropriate front-end devices along with a list of clients to query. The back-end

device will collect answers and coins, and add noise. As with front-end devices, the back-end devices do not require interaction beyond redundancy needs, and so also scale with the addition of more devices.

4 Privacy Analysis

In this section, we provide an analysis of the privacy properties of PDDP design given the threat assumptions from §2.

4.1 Analyst

The analyst can influence the content of the messages that it originates (i.e., "$query, \{L_i, U_i\}_{i=1}^{b}, c, \varepsilon$" in Step 1), and the messages that infected clients originate (i.e., "$\{v_i\}_{i=1}^{b}$" in Step 3).

4.1.1 Defending Against Malicious Analyst

An analyst could generate a query that attempts to reveal an individual client's information, for instance by specifying personally identifiable information (PII) as a predicate in the query. The noise added by the proxy defeats this (Step 4), at least within the guarantees given by differential privacy. Note that in most application domains, it may be perfectly reasonable to disallow PII as a predicate, thus giving stronger privacy protection in practice.

By enforcing a maximum limit of ε, the proxy prevents an analyst from giving a large ε in the query to minimize the added noise (Step 2). The analyst may select a small value of c in an attempt to isolate a single client. The differentially private noise defeats this (Step 4). Alternatively, the proxy can also simply enforce a lower bound on the value of c. In addition, the fact that the proxy selects random clients to query prevents an analyst from knowing which clients answered (policy 1 in Step 2). The session encryption between proxy and clients prevents an eavesdropping analyst from knowing which client received which query (§3.2.1). If necessary, the proxy can add delay or chaff to the queries sent to clients, in order to prevent an eavesdropping analyst from identifying the queried clients by using traffic analysis to correlate the query sent to the proxy with the subsequent messages sent to clients.

On the assumption that clients have the proxy's correct public key, the session encryption prevents the analyst from becoming a man in the middle between clients and proxy (§3.2.1). Similarly, the analyst as an eavesdropper cannot see the answers generated by the clients, and so cannot trivially decrypt them and associate answers with clients. The analyst also cannot, through traffic analysis, correlate answers received by the proxy to those sent by the proxy. This is because the proxy stores, delays, and shuffles all answers and coins before sending them to the analyst (Step 5).

4.1.2 Defending Against Clients Infected by Analyst

The analyst may try to create a covert channel between itself and infected clients by manipulating client answers. However, the analyst cannot create a covert channel by having the infected client directly embed it into the transmitted ciphertext. This is because the resulting ciphertext would not satisfy the Jacobi symbol checking made by the proxy (Step 3), and so would be discarded. The analyst also cannot create a covert channel by creating a word from per-bucket values in the full answer $\{v_i\}_{i=1}^{b}$. This is because all the answer and noise values within each individual bucket are mixed and shuffled at the proxy (Step 5). Note that this also prevents the identification of an individual client through correlation across per-bucket answer values.

In the GM bit-cryptosystem, the encryption algorithm $\mathscr{E}$ takes one bit $w \in \{0,1\}$ and the public key (x,m) as input, and outputs the ciphertext $\mathscr{E}(w) = r^2x^w \pmod{m}$, where r is a random number from $\mathbb{Z}_m^*$. Without the fix introduced at the end of this paragraph, the analyst could create a covert channel by knowing the sequence of random numbers r that a given client uses to produce the ciphertexts. This could be done for instance by having the client use the same known random number for each answer bit, or by knowing the seed for the random number generator. This would allow the analyst to predict which ciphertext would be produced for the subsequent '1' or '0' from the client, thus allowing it to isolate that client's answer stream from those of other clients. This covert channel can easily be destroyed by having the proxy *re-randomize* every client-produced answer value by homomorphically XOR-ing it with a randomly encrypted '0'. This effectively scrambles the ciphertext without modifying the client's answer.

4.2 Client

An adversary may try to distort the final aggregate answer by creating clients that produce incorrect answers. Because of the use of GM bit-cryptosystem, individual clients are strictly limited to generate only a single bit per bucket. Therefore, the *maximum* absolute distortion per bucket in the final aggregate answer is bounded by the number of malicious clients (here ignoring distortion due to the differentially private noise). This is in contrast with previous distributed differential privacy designs [30, 45, 49], where even a *single* malicious client can substantially distort the final aggregate answer.

Our system thus raises the bar for the adversary by forcing it to deploy a large number of malicious clients,

especially when limits are placed on the number of clients that can answer a query from a single IP address, as described in §3.3.4. Deploying a large number of malicious clients increases the likelihood that the nature of the client attack will be detected, thus allowing the analyst to at least know which queries are incorrect. What's more, even when there are a large number of malicious clients, due to the noise added by the proxy, our system *always* provides differentially private guarantees at least for every honest client.

4.3 Proxy

There are many legitimate cases where an analyst may wish to publish the (noisy) aggregate answers that it learns. Because the honest-but-curious proxy does not know the actual value of the coins that it uses to add noise (see §3.2.3), it cannot determine the true noise-free aggregate answer by subtracting the added noise from the aggregate answer published by the analyst.

Note that it would be trivial for a *dishonest* proxy to generate the required unbiased coins by itself (i.e., not base them on a re-flipping of the client-generated coins). This could not be detected because the correctly generated coins in any event appear as random perturbations on the original client-generated coins. Were this dishonest proxy a concern in practice, we could require that the analyst itself add differentially private noise to final aggregate answers before publishing them. Of course, a malicious analyst may choose not to add this noise. Absent collusion, however, the proxy would not know which analysts were adding noise, and which were not.

5 Implementation and Evaluation

5.1 Implementation and Deployment

We implemented a fully functional PDDP system. Following our design in §3.2, the implementation comprises three basic components: client, proxy, and analyst.

The **client** is implemented as a Firefox add-on (as shown in Figure 3). It consists of 9600 lines of Java code, compiled into JavaScript using the Google Web Toolkit. It captures users' web browsing activities (such as webpages visited, searches made, etc.), certain online shopping activities (such as how often items are placed in shopping carts), and certain ad interactions (such as the number of ads viewed and clicked). In principle, our Firefox add-on can be extended to capture any online activities made by the user. All captured information is stored in a local SQLite database using Firefox's Storage API, thus allowing our system to query for that information. Every 5 minutes, the add-on connects to the proxy

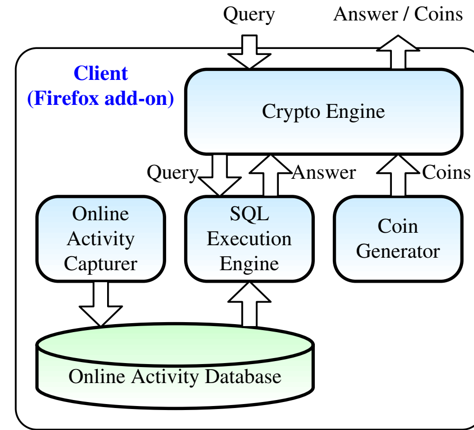

Figure 3: Client implementation

to retrieve any pending queries, and return answers and periodically generated coins.

The **proxy** is implemented with 3600 lines of code as a web server running Tomcat 6.0.33. It forwards the queries from analyst to clients, receives client-generated answers and coins, adds differentially private noise, and sends answers with noise back to the analyst. Proxy state is stored in a back-end MySQL database.

The **analyst** is implemented in 800 lines of Java code.

After verifying the correctness of PDDP on a set of local machines (where we have access to the local data), we deployed PDDP on more than 600 actual clients. This deployment allows us to make queries to exercise and test our system.

5.2 Comparison: An Alternative Design

In our evaluation, we compare PDDP with a strawman design that is more straightforward and might at first glance appear to be a natural choice. Indeed, this was one of the designs that we initially focused on. Like [45], it exploits the additive homomorphism of the Paillier cryptosystem [42]. Like PDDP, however, it uses an honest-but-curious proxy in order to avoid expensive protocols like the distributed key distribution in [45].

In the "Paillier-based" design, each queried client returns an encrypted 1 or 0 in each bucket. Due to Paillier's additive homomorphism, the proxy can directly sum up all clients' encrypted answers to get the encrypted total sum for each bucket. The proxy adds the required differentially private noise to each bucket to generate the encrypted noisy aggregate answer, and then forwards this answer to the analyst. The analyst uses its private key to decrypt the noisy aggregate answer.

Like [45], this basic design suffers from the problem that a single malicious client can provide an answer value other than 1 or 0 in each bucket to substantially distort the

Table 1: Performance at client (sustained crypto operations per second)

	Encryption			ZKP Generation		
	Firefox	Chrome	Smartphone	Firefox	Chrome	Smartphone
PDDP System	2157.96	22773.86	808.87	–	–	–
Paillier-based System	0.59	7.79	0.27	0.17	2.34	0.08

Table 2: Performance at proxy and analyst (sustained crypto operations per second). Homomorphic operations are XOR in the PDDP system, and addition in the Paillier-based system.

	Encryption	Decryption	Homomorphic Op	ZKP Generation	ZKP Verification
PDDP System	15323.32	6601.10	123609.39	–	–
Paillier-based System	62.76	63.41	34188.03	18.70	15.58

aggregate answer without detection. To overcome this problem, we used non-interactive zero-knowledge proofs (ZKP) [8] based on the Fiat-Shamir heuristic [23] to ensure that all encrypted answer values are either 1 or 0. The resulting design is almost apples-to-apples comparable with PDDP. The primary difference, however, is that in the Paillier-based design, the proxy knows exactly how much noise has been added, and therefore knows the true aggregate answer in those cases where the analyst publishes its noisy aggregate answer (see §4.3).

5.3 Evaluation

5.3.1 Computational Overhead

A major concern in the performance of distributed differentially private systems is the overhead of the asymmetric crypto operations. In this section, we measure the crypto performance of PDDP, as well as compare it with the Paillier-based system outlined in §5.2.

To generate an answer to a received query, each client needs to encrypt a binary value for each bucket, using the GM cryptosystem in PDDP or using the Paillier cryptosystem in a Paillier-based system. We measure the sustained rate at which three different clients can do these crypto operations: a Firefox browser and a Chrome browser on a desktop workstation running Linux 2.6.32 with Intel dual core 3GHz, and a WebKit browser on a smartphone running Android 2.2 with a 1GHz processor. The results, with a 1024-bit key length, are given in Table 1. The PDDP crypto operations are very fast. Even the smartphone can execute over 800 encryptions per second. This suggests that crypto is not a bottleneck in PDDP's client implementation, though in practice these operations should be run in the background for queries with many buckets. By contrast, the Paillier-based clients are prohibitively slow: 8 encryptions per second with Chrome, and less than one per second for the smartphone. If zero-knowledge proofs (ZKPs) are used to ensure that clients produce answers with only 1's and 0's (see §5.2), Paillier-based clients are even slower.

At the proxy, the PDDP system needs to perform GM crypto-operations to transform periodically client-generated coins into unbiased coins — one (offline) encryption and one homomorphic XOR for one unbiased coin. Moreover, the PDDP proxy needs to perform the Jacobi symbol checking on received coins and answer values. This checking cost is the same as (or faster than) the cost of GM decryption. Table 2, also with a 1024-bit key length, shows that all these crypto operations are quite efficient in PDDP. In a Paillier-based system, the proxy needs to first verify all clients' submitted ZKPs — one ZKP for each client answer's each bucket. The ZKP verification is very expensive as shown in Table 2. Furthermore, the proxy in a Paillier-based system needs to homomorphically sum up all client answers for each bucket, and then add locally generated differentially private noise to each per-bucket total sum.

To give a concrete example, we assume a 10-bucket query sent to 1M clients, with a conservative privacy parameter of $\varepsilon = 1.0$ [39]. Based on the results in Table 2, the PDDP proxy would require less than 1 CPU-second for the GM crypto-operations, and less than 30 CPU-minutes for the Jacobi symbol checking. The Paillier-based proxy, by contrast, would require roughly 5 CPU-minutes for the homomorphic additions, and roughly 1 CPU-week for the zero-knowledge proof verifications.

At the analyst, while the PDDP system needs to decrypt all encrypted values (i.e., all client answers plus coins) within each bucket, the Paillier-based system only needs to decrypt one encrypted value within each bucket. The computational overhead at the PDDP analyst is higher than at the Paillier-based analyst. Nevertheless, the overhead is very reasonable considering the efficiency of PDDP's crypto operations.

5.3.2 Bandwidth and Storage Overhead

In both PDDP and Paillier-based systems, a client needs to transmit an encrypted answer value for each bucket. A

PDDP client also needs to transmit a periodically generated coin to the proxy, while a Paillier-based client needs to transmit a ZKP for each bucket of an answer. The bandwidth requirements for these are modest: on the order of a few kilobytes.

The PDDP proxy needs to store all queried clients' encrypted answer values for each bucket. Assuming a key length of 1024 bits, the storage for a 10-bucket query over 1 million clients is about 1.2GB. These answers can be stored on a hard-drive. The PDDP proxy also needs to receive, re-flip, and store client-generated coins. The required number of coins, however, is only a small fraction of the number of answer values. As an example, for the 10-bucket, 1M-client, $\varepsilon = 1.0$ query, only 9290 coins are required (929 per bucket). Finally, the answers and coins will be transmitted to the analyst (roughly 1.2GB), which stores and decrypts them. Overall, the storage and bandwidth requirements for PDDP proxy are quite reasonable. Recall that it is straightforward to further scale the system by adding proxy devices (see §3.3.5).

Note that the Paillier-based proxy and analyst have minimal storage and bandwidth requirements because they need to store and transmit only one aggregate answer value per bucket. Each time the Paillier-based proxy receives an answer from a client, it adds the per-bucket answer value to the corresponding bucket of the locally maintained aggregate answer.

In summary, the PDDP system overhead is within very reasonable limits for all system components. Compared with a Paillier-based system, the PDDP system trades-off moderate and affordable bandwidth and storage overhead against unacceptable computational overhead.

5.3.3 Querying the Client Deployment

Before we deployed PDDP at scale, we first tested it on a set of local machines. Since we had access to the data on the local machines, we knew the true answers of the clients running on those local machines. In so doing, we were able to verify the correctness of PDDP. We then deployed PDDP on more than 600 clients taken from "friends and family" as well as Amazon Mechanical Turk. This deployment allowed us to make queries to exercise and test the PDDP system.

Figure 4 gives a histogram of the number of connections received from clients that established at least one connection on Sep. 26. While the Firefox browser is active, clients make connections every 5 minutes. We see that only one or two percent of client browsers were running the full 24 hours. Almost half of the clients made 20 or fewer connections, representing a browser uptime of less than 2 hours in 24. This suggests that it could easily take several days to complete a query over nearly all of a set of clients. It also suggests that an approach

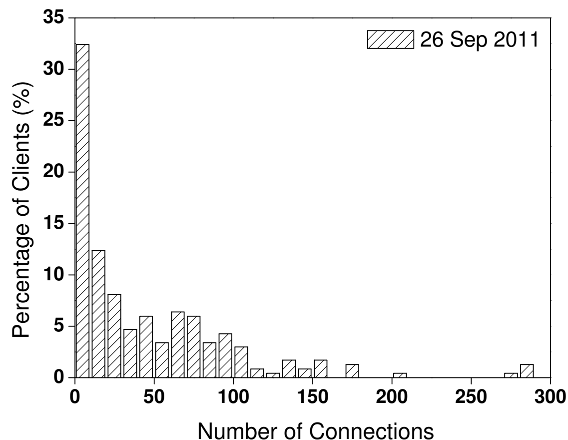

Figure 4: Number of connections received from clients on Sep. 26, 2011

that requires that clients are continuously available, as with [45], will not work in our setting.

To further test our system, we executed a number of PDDP queries over our deployed client base. Each query sets $c = 250$ (out of over 600 clients) and $\varepsilon = 5.0$. Each query lasts 24 hours, and we select clients as they connect until $c = 250$ unique clients are queried or 24 hours expires, whichever comes first. Finally, each query covers statistics gathered over a 24 hour period — midnight-to-midnight of the day specified in the query in the client's local time.

Note that what we report here are the noisy aggregate answers from our deployment. To preserve users' privacy, PDDP didn't and indeed shouldn't output the true noise-free aggregate answers. This is our key premise and motivation — learn statistical results without knowing users' true answers. Nevertheless, our experimental parameter settings (i.e., $c = 250$ and $\varepsilon = 5.0$) result in 16 coins per bucket, and ensure that a per-bucket aggregate answer is within plus or minus 2, 4, and 6 of the true noise-free aggregate answer, with the probability of 68%, 95%, and 99.7%, respectively (see §3.3.1).

We started by issuing the simple query "how many webpages did a client visit per day?" over five consecutive days. We specified 8 buckets as shown in Table 3. Between 176 and 230 answers were gathered each day (including noise). Since each client supplies a count in a single bucket only, this directly reflects (noisily) the number of unique clients that replied in 24 hours. The results in Table 3 show that the requested information can usefully be gathered through the PDDP system. For instance, we see that a large number of clients show very little activity (0$\sim$10 webpages visited). What's more, the number of these low-activity clients was greater during the weekend (Sep. 24 and 25). On the other hand, a

Table 3: Number of webpages visited per day (displayed as noisy client counts within each bucket)

	0∼10 pages	11∼20 pages	21∼50 pages	51∼100 pages	101∼200 pages	201∼500 pages	501∼1000 pages	>1000 pages	Total
24 Sep 2011	73	1	10	14	29	33	10	6	176
25 Sep 2011	115	3	14	11	30	43	17	-3	230
26 Sep 2011	61	20	16	21	44	48	12	3	225
27 Sep 2011	49	8	15	17	25	44	20	2	180
28 Sep 2011	56	7	8	19	27	45	16	3	181

Table 4: Five different queries towards clients' activities on Sep. 29, 2011 (displayed as noisy client counts within each bucket)

	0	1∼10	11∼20	21∼50	51∼100	101∼200	201∼500	501∼1000	>1000	Total
Q_1	45	11	5	18	23	24	37	18	4	185
Q_2	46	40	28	45	19	7	0	0	-3	182
Q_3	43	17	9	23	24	30	33	2	2	183
Q_4	56	32	20	28	31	8	6	-1	2	182
Q_5	77	28	17	25	8	16	6	5	1	183

Q_1: How many webpages visited?
Q_2: How many unique websites visited?
Q_3: How many visited webpages on a user's top 3 favorite websites?
Q_4: How many searches made?
Q_5: How many Google ads shown?

substantial number of clients visit between 100 and 500 webpages per day. Note that in one case the added noise produces a negative count. This poses no privacy risk; if desired, an analyst can safely replace all negative counts with zeros without privacy loss [33].

We additionally issued five queries towards clients' activities on Sep. 29, as listed in Table 4. We also added a '0' bucket in order to distinguish between truly idle clients and low-activity clients. Once again, this set of queries shows that it is possible to gather meaningful noisy data from even a relatively small population of clients. For instance, we see from query Q_1 that indeed a large fraction of low-activity browsers are idle (45 clients in '0' bucket). We see from query Q_2 that a relatively large number of users (40 of 136 non-idle clients) visited a small number of unique websites (between 1 and 10). Moreover, given the rough similarity between the results of queries Q_1 and Q_3, it appears that most browsing takes place over users' top 3 favorite websites. Many additional observations can be made; for instance, from queries Q_4 and Q_5 we see that search is often used (around half of the clients made 11∼100 searches), and that Google ads are shown relatively often.

6 Malicious Proxy

In this section, we sketch out how we may weaken proxy trust requirements through the use of trusted hardware such as the Trusted Platform Module (TPM) or the IBM 4758 cryptographic coprocessor [44]. In particular, we assume that the proxy is malicious in that it may try to violate the privacy of individual clients, and is willing to collude with analysts (or pose as an analyst). On the other hand, we assume that the proxy is unwilling or unable to physically tamper with the hardware, including placing probes on the system bus in order to record system operations, for instance because there is a threat of random inspections from a third party.

The basic idea is that the proxy runs an executable which has been verified by a trusted third party to operate correctly as a proxy. This executable is then remotely attested by the clients, as described in [44].

There are two attacks that we need to defend against. First, the proxy may try to de-anonymize a client by associating a given answer with the client's IP address, and conveying this association to the analyst. Second, the proxy may try to avoid adding noise to the answers, thus violating differential privacy. We assume that the proxy operator is able to launch a reboot attack on the proxy hardware. In this attack, after a client attests the executable and transmits its answer to the proxy, the proxy is rebooted with a malicious executable which carries out one of the above attacks. Prior work has shown how to prevent a reboot attack once the client maintains a secure channel with the proxy after attestation [25, 37]. In our scenario, however, a client may drop the secure channel after it has supplied its answer (Step 3) but before the proxy transmits the answer to the analyst (Step 5).

In our design, we assume that after a client connects to the proxy, the query is received and answered, as well as any coins are delivered, within the same attested secure channel. The proxy operates as follows:

- When the proxy receives a coin from a client, it re-flips the coin and stores it into the unbiased coin pool in memory. Upon booting, previous stored coins will be lost and the pool will be empty. Therefore, the proxy will have to gather a modest number of coins (some thousands) before it can start handling queries from analysts.
- When the proxy receives a query from the analyst, it immediately reads the required number of coins from the unbiased coin pool, and places them in a newly initialized linked list or similar data structure. If there are not enough unbiased coins remaining in the pool, the proxy waits until enough coins have been generated.
- When the proxy receives a (periodic) connect request from a client with a pending query, it sends the query to the client and receives an answer. The answer is placed in the linked list at a random location, and no association with the client answer is established.
- When enough client answers have been received, all items in the linked list (answers and coins mixed together) are transmitted to the analyst.

Analysis. By requiring that the proxy read the coins before receiving any client answers, we ensure that even a single client answer is adequately mixed with noise. This prevents a client de-anonymization attack whereby the proxy is rebooted with a malicious executable after only a single answer is received, thus allowing the malicious executable to link the received answer to the client by recording network activity. This also prevents the no-noise attack, by ensuring that noise is mixed in from the very beginning, and cannot be distinguished from client answers. Further work is required to fully analyze this approach to accommodating a malicious proxy.

7 Related Work

To preserve user privacy, early approaches like anonymization sanitize the user data by removing well-known personally identifiable information (PII), e.g., name, gender, birthday, social security number, ZIP code, etc. These approaches not only heavily restrict the utility of the user data [43], but ultimately cannot effectively protect user privacy [11, 41, 46]. In a well-publicized example of this, an individual user was identified from AOL's anonymized search logs [1]. A number of privacy-preserving approaches have been proposed [5, 6, 21, 22, 35, 36, 50], discussed in the following.

One general approach [5, 6] is to randomize user data by adding random distortion values drawn independently from some known distribution such as a uniform distribution or a normal distribution. An analyst can collect the distorted user data and reconstruct the distribution of the original data using for instance the expectation maximization (EM) algorithm [5]. However, [22] indicates that such simple randomization is potentially vulnerable to privacy breaches. Evfimievski *et al.* then proposed a class of new randomization operators [22], and used them to limit the privacy breaches [21].

k-anonymity [50] ensures that each individual is indistinguishable from at least $k-1$ other individuals with respect to certain sensitive attributes, thus individuals cannot be uniquely identified. While k-anonymity protects against identity disclosure, it does not prevent attribute disclosure if there is little diversity in these sensitive attributes. To address this problem, l-diversity [36] was proposed which requires that there exist at least l well-represented values for each sensitive attribute. Recently, [35] indicated that l-diversity may not be necessary, and it is also insufficient to prevent attribute disclosure. To solve the problems of both k-anonymity and l-diversity, [35] further proposed t-closeness which requires that the distribution of a sensitive attribute in any "equivalence class" is close to the overall distribution of the attribute, and the distance between the two distributions is no more than a threshold t.

Many of these approaches, however, only provide syntactic guarantees on the privacy, and impose constraints of one sort or another. Differential privacy [13, 14, 17] is considered stronger than previous approaches because it provides a provable guarantee that it is hard to detect the presence or absence of any individual records, as already discussed in §3.1. Differential privacy does not make any assumptions about the adversary. It is independent of the adversary's computational power and auxiliary information. However, original differential privacy mechanisms were not designed to support a distributed environment.

In theory, secure multi-player computation [26, 52] could be used to emulate differential privacy in a distributed manner. It would be, however, highly expensive. To our knowledge, there are four previous designs for distributed differential privacy [16, 30, 45, 49]. None of them are realistically practical in a large-scale distributed setting.

The first distributed differential privacy design was proposed by Dwork *et al.* in [16]. It is something of a cryptographic tour de force, variously exploiting Byzantine agreement, distributed coin flipping, verifiable se-

cret sharing, secure function evaluation, and randomness extractor. However, its computational load per user is $O(U)$, where U is the number of users, making it impractical for a large-scale setting.

To reduce this complexity, Rastogi and Nath in [45] designed a two-round protocol based on the threshold Paillier cryptosystem, and Shi *et al.* in [49] designed a single-round protocol based on the decisional Diffie-Hellman assumption. Both designs [45, 49] achieve distributed differential privacy while reducing the computational load per user from $O(U)$ to $O(1)$. However, their distributed key distribution protocols cannot work at large-scale under churn. To solve this problem, Götz and Nath in [30] utilized two honest-but-curious servers to collaboratively collect users' noisy answers and compute the final aggregate result. Nevertheless, in all of these designs [30, 45, 49], even a single malicious user can substantially distort the final result without detection.

8 Conclusion and Future Work

In presenting a practical system, PDDP, that supports statistical queries over distributed private user data, this paper takes a first step towards practically mining that data while preserving privacy. Still, there is a gap between the utility of a central database and that of PDDP. While one might be willing to give up some utility in exchange for privacy, it is important to maximize the utility of distributed private user data.

Previous work shows that it is possible to support various statistical learning algorithms through a differentially private interface [9, 10]. These algorithms, however, require a sequence of queries to a given database (or, in our case, a given set of clients). It is not clear whether this is practical over PDDP. While in principal the proxy may retain state about which set of clients should receive a sequence of queries, in practice some fraction of these clients will become unavailable temporarily or permanently during the course of the queries. One avenue of future work is to understand the impact of this on the utility of the data as well as the time it takes to run these algorithms.

A second area of future work concerns the scalability of non-numeric queries (§3.3.3). A brute-force approach of assigning a numeric value to every possible non-numeric answer does not scale for certain queries, e.g., "what is the most frequent search?". One possible approach might be to use invertible Bloom filters [20, 29] to map a large number of possible non-numeric answers into a relatively small number of buckets, at some loss of fidelity. However, we need to understand the trade-off between utility and scalability. An interesting question is whether this loss of fidelity itself can be exploited as a privacy mechanism.

More broadly, we need to gain experience with PDDP. Towards this end, we plan to use PDDP to gather statistical data for a large-scale experiment with the Privad privacy-preserving advertising system [31]. By "eating our own dog food", we can learn first-hand the limitations of PDDP in gathering meaningful data from clients.

While we believe that it is reasonable in practice to require an honest-but-curious proxy, it would obviously be better not to. One avenue of future work is to nail down the design and security properties of a TPM-based approach as sketched in §6. Ultimately, a challenging open problem is to design a system that practically provides differentially private statistical queries over distributed data without requiring a trusted third party.

Finally, the question of measuring actual privacy loss (versus worst-case privacy loss) is extremely important for differential privacy, not only in a distributed setting but generally. While we assume that this necessarily involves incorporating domain-specific information such as user sensitivity and adversarial knowledge, and therefore gives up some of the rigor of differential privacy, we believe it is important to make progress here in order to make differential privacy more practical.

9 Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Nickolai Zeldovich, for their insightful comments. We also thank Rose Hoberman, Aniket Kate, Stevens Le Blond, and Nan Zheng for their valuable feedback on drafts of this work. Finally, we would like to thank the volunteers for anonymously helping us exercise our system.

References

[1] A Face Is Exposed for AOL Searcher No. 4417749. http://www.nytimes.com/2006/08/09/technology/09aol.html.

[2] FTC Charges Deceptive Privacy Practices in Google's Rollout of Its Buzz Social Network. http://www.ftc.gov/opa/2011/03/google.shtm.

[3] Microsoft HealthVault. http://www.microsoft.com/en-us/healthvault/.

[4] Project VRM. http://cyber.law.harvard.edu/projectvrm/Main_Page.

[5] AGRAWAL, D., AND AGGARWAL, C. C. On the Design and Quantification of Privacy Preserving Data Mining Algorithms. In *PODS* (2001).

[6] AGRAWAL, R., AND SRIKANT, R. Privacy-Preserving Data Mining. In *SIGMOD Conference* (2000), pp. 439–450.

[7] BACKES, M., KATE, A., MAFFEI, M., AND PECINA, K. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *IEEE Symposium on Security and Privacy* (2012).

[8] BAUDRON, O., FOUQUE, P.-A., POINTCHEVAL, D., STERN, J., AND POUPARD, G. Practical multi-candidate election system. In *PODC* (2001), pp. 274–283.

[9] Blum, A., Dwork, C., McSherry, F., and Nissim, K. Practical privacy: the SuLQ framework. In *PODS* (2005), pp. 128–138.

[10] Blum, A., Ligett, K., and Roth, A. A learning theory approach to non-interactive database privacy. In *STOC* (2008), pp. 609–618.

[11] Coull, S. E., Wright, C. V., Monrose, F., Collins, M. P., and Reiter, M. K. Playing Devil's Advocate: Inferring Sensitive Information from Anonymized Network Traces. In *NDSS* (2007).

[12] Douceur, J. R. The Sybil Attack. In *IPTPS* (2002), pp. 251–260.

[13] Dwork, C. Differential Privacy. In *ICALP* (2006), pp. 1–12.

[14] Dwork, C. Differential Privacy: A Survey of Results. In *TAMC* (2008), pp. 1–19.

[15] Dwork, C. A firm foundation for private data analysis. *Commun. ACM 54*, 1 (2011), 86–95.

[16] Dwork, C., Kenthapadi, K., McSherry, F., Mironov, I., and Naor, M. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *EUROCRYPT* (2006), pp. 486–503.

[17] Dwork, C., McSherry, F., Nissim, K., and Smith, A. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC* (2006), pp. 265–284.

[18] Eikenberry, S. M., and Sorenson, J. Efficient Algorithms for Computing the Jacobi Symbol. In *ANTS* (1996), pp. 225–239.

[19] Enck, W., Gilbert, P., gon Chun, B., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI* (2010), pp. 393–407.

[20] Eppstein, D., and Goodrich, M. T. Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters. *IEEE Trans. Knowl. Data Eng. 23*, 2 (2011), 297–306.

[21] Evfimievski, A. V., Gehrke, J., and Srikant, R. Limiting privacy breaches in privacy preserving data mining. In *PODS* (2003), pp. 211–222.

[22] Evfimievski, A. V., Srikant, R., Agrawal, R., and Gehrke, J. Privacy preserving mining of association rules. In *KDD* (2002), pp. 217–228.

[23] Fiat, A., and Shamir, A. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO* (1986), pp. 186–194.

[24] Ghosh, A., and Roth, A. Selling privacy at auction. In *ACM Conference on Electronic Commerce* (2011), pp. 199–208.

[25] Goldman, K., Perez, R., and Sailer, R. Linking remote attestation to secure tunnel endpoints. In *STC* (2006), pp. 21–24.

[26] Goldreich, O., Micali, S., and Wigderson, A. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC* (1987), pp. 218–229.

[27] Goldwasser, S., and Micali, S. Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information. In *STOC* (1982), pp. 365–377.

[28] Goldwasser, S., and Micali, S. Probabilistic Encryption. *J. Comput. Syst. Sci. 28*, 2 (1984), 270–299.

[29] Goodrich, M. T., and Mitzenmacher, M. Invertible Bloom Lookup Tables. *CoRR abs/1101.2245* (2011).

[30] Götz, M., and Nath, S. Privacy-Aware Personalization for Mobile Advertising. In *Microsoft Research Technical Report MSR-TR-2011-92* (2011).

[31] Guha, S., Cheng, B., and Francis, P. Privad: Practical Privacy in Online Advertising. In *NSDI* (2011).

[32] Haeberlen, A., Pierce, B. C., and Narayan, A. Differential privacy under fire. In *USENIX Security Symposium* (2011).

[33] Korolova, A., Kenthapadi, K., Mishra, N., and Ntoulas, A. Releasing search queries and clicks privately. In *WWW* (2009), pp. 171–180.

[34] Krishnamurthy, B., and Wills, C. E. On the leakage of personally identifiable information via online social networks. In *WOSN* (2009), pp. 7–12.

[35] Li, N., Li, T., and Venkatasubramanian, S. t-Closeness: Privacy Beyond k-Anonymity and l-Diversity. In *ICDE* (2007), pp. 106–115.

[36] Machanavajjhala, A., Gehrke, J., Kifer, D., and Venkitasubramaniam, M. l-Diversity: Privacy Beyond k-Anonymity. In *ICDE* (2006).

[37] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: an execution infrastructure for tcb minimization. In *EuroSys* (2008), pp. 315–328.

[38] McSherry, F. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD Conference* (2009), pp. 19–30.

[39] McSherry, F., and Mahajan, R. Differentially-private network trace analysis. In *SIGCOMM* (2010), pp. 123–134.

[40] McSherry, F., and Mironov, I. Differentially Private Recommender Systems: Building Privacy into the Netflix Prize Contenders. In *KDD* (2009), pp. 627–636.

[41] Narayanan, A., and Shmatikov, V. Robust De-anonymization of Large Sparse Datasets. In *IEEE Symposium on Security and Privacy* (2008), pp. 111–125.

[42] Paillier, P. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT* (1999), pp. 223–238.

[43] Pang, R., Allman, M., Paxson, V., and Lee, J. The devil and packet trace anonymization. *Computer Communication Review 36*, 1 (2006), 29–38.

[44] Parno, B., McCune, J. M., and Perrig, A. Bootstrapping Trust in Commodity Computers. In *IEEE Symposium on Security and Privacy* (2010), pp. 414–429.

[45] Rastogi, V., and Nath, S. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD Conference* (2010), pp. 735–746.

[46] Ribeiro, B. F., Chen, W., Miklau, G., and Towsley, D. F. Analyzing Privacy in Enterprise Packet Trace Anonymization. In *NDSS* (2008).

[47] Roy, I., Setty, S. T. V., Kilzer, A., Shmatikov, V., and Witchel, E. Airavat: Security and Privacy for MapReduce. In *NSDI* (2010), pp. 297–312.

[48] Shallit, J., and Sorenson, J. A binary algorithm for the Jacobi symbol. *ACM SIGSAM Bulletin 27*, 1 (1993), 4–11.

[49] Shi, E., Chan, T.-H. H., Rieffel, E. G., Chow, R., and Song, D. Privacy-Preserving Aggregation of Time-Series Data. In *NDSS* (2011).

[50] Sweeney, L. k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10*, 5 (2002), 557–570.

[51] Toubiana, V., Narayanan, A., Boneh, D., Nissenbaum, H., and Barocas, S. Adnostic: Privacy Preserving Targeted Advertising. In *NDSS* (2010).

[52] Yao, A. C.-C. Protocols for Secure Computations. In *FOCS* (1982), pp. 160–164.