SplitX: High-Performance Private Analytics

Ruichuan Chen
Bell Labs / Alcatel-Lucent
ruichuan.chen@alcatel-lucent.com

Istemi Ekin Akkus
MPI-SWS
iakkus@mpi-sws.org

Paul Francis
MPI-SWS
francis@mpi-sws.org

ABSTRACT

There is a growing body of research on mechanisms for preserving online user privacy while still allowing aggregate queries over private user data. A common approach is to store user data at users' devices, and to query the data in such a way that a differentially private noisy result is produced without exposing individual user data to any system component. A particular challenge is to design a system that scales well while limiting how much the malicious users can distort the result. This paper presents the design of *SplitX*, a high-performance analytics system for making differentially private queries over distributed user data. SplitX is typically two to three orders of magnitude more efficient in bandwidth, and from three to five orders of magnitude more efficient in computation than previous comparable systems, while operating under a similar trust model. SplitX accomplishes this performance by replacing public-key operations with exclusive-or operations. This paper presents the design of SplitX, analyzes its security and performance, and describes its implementation and deployment across 416 users.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Analytics; differential privacy; XOR cryptography

1. INTRODUCTION

The tracking of online users has become a major concern, leading to both government and industry regulations to limit tracking, e.g., the EU Cookie Law [2] and the W3C Do-Not-Track [3]. The two primary use cases for tracking are behavioral advertising and analytics. Unfortunately, regulations like the EU Cookie Law and Do-Not-Track are only able to limit tracking at the expense of advertising and analytics.

SIGCOMM'13, August 12–16, 2013, Hong Kong, China.

Concurrent to these developments, there has been active research on ways to allow behavioral advertising [6,15,19,32] or analytics [4,9,13,20,26,28] while still protecting user privacy. An approach that satisfies both industry and consumers is better than the current forced compromise [7]. Unfortunately, to our knowledge, none of the proposals cited above have made a noticeable impact on commercially deployed systems. Regarding the research on private analytics systems, which is the topic of this paper, we believe that all of the previous proposals have technical shortcomings that make them unsuitable for commercial acceptance.

A common requirement among previous analytics proposals is that *no* system component should ever have access to individual user data. (This requirement is in contrast to most database privacy research, which usually assumes that there is a trusted database that sees all user data, e.g., hospital records or records of online purchases.) To satisfy this requirement, all the previous proposals assume that user data resides on users' own devices or devices the users trust. In other words, user data is distributed across a potentially large number of *client* devices.

A common approach among previous proposals is that they all add *differentially private* noise [11, 12, 14] to the aggregate result of queries, so that the privacy of any individual user is protected. The query answers transmitted by client devices flow through one or more servers that obliviously generate answer aggregates and output the noisy aggregates. By "oblivious", we mean that the servers aggregate answers without seeing individual user answers. Within this framework, there are two places where differentially private noise can be added: either the client adds noise to the answer it is transmitting [13, 20, 26, 28], or the servers *blindly* add noise such that they do not know how much noise is added [4, 9]. It is important that servers add noise blindly; otherwise, they could determine the noise-free aggregate if, for instance, the noisy aggregate were publicly released.

There are two primary technical shortcomings of the previous analytics proposals that make them unsuitable for commercial use: *scalability* and *answer pollution*. Regarding answer pollution, three of the proposals [20,26,28] where the clients add noise suffer from the problem that even a *single* malicious client can substantially pollute the aggregate result through the transmission of a *single* answer. This is unacceptable in a world where any user may be malicious, or any user device may be corrupted with malware. Regarding scalability, all of the proposals, except Hardt et al. [20], require at least public-key operations or something even more expensive. For instance, PDDP [9] and Akkus et al. [4], both

of which scale substantially better than any of [13, 26, 28], nevertheless require a public-key operation per *single* bit of client answer, and require roughly one kilobit of message data per *single* bit of client answer.

Hardt et al.'s proposal [20], by contrast, scales much better, requiring only a few arithmetic operations per client answer, and only roughly two bits of message data per bit of client answer. Besides being susceptible to answer pollution attacks, however, this system is also not very general. It is designed on the premise that the clients know a priori what queries to answer, and that all clients answer the same set of queries. In a general setting where different *analysts* wish to query different communities of clients, this system does not scale.

This paper presents *SplitX*, a system for making differentially private queries over distributed user data that scales substantially better than previous systems, does not suffer from answer pollution attacks, and is general. SplitX operates under a similar trust model as several recent analytics proposals [4, 9, 20], i.e., honest-but-curious servers that do not collude, and untrusted analysts.

SplitX derives its scalability from the fact that it uses a simple exclusive-or (XOR) operation as its crypto primitive. The sending client XOR's its message with a random string of identical length. It sends the XOR'ed message via one proxy, and the random string via another proxy, thus effectively producing a one-time pad. As a bandwidth optimization, the randomly selected seed, used as the input to the pseudo-random number generator (PRNG), may be used in lieu of the random string.

SplitX is resistant to malicious clients attempting to pollute answers. Any single answer can only modify an aggregate answer by a single count. In addition, SplitX has mechanisms to detect whether a client, as identified by its IP address, repeatedly answers the same query.

SplitX is a general system in that an untrusted analyst can direct its own queries (e.g., SQL queries with counts as answers) to a selected (but anonymous) client population, and receive differentially private results. SplitX anonymously distributes queries to and receives answers from only the selected clients. For instance, an analyst that wants to query only clients that have visited a specific website may do so without having to send the query to all clients. This is accomplished through a kind of anonymous pub-sub channel: clients that, for instance, visit a given website "subscribe" to a channel associated with visitors of the website, and analysts interested in that set of clients "publish" queries to that channel.

Altogether, this paper makes the following contributions:

- It presents the design of SplitX, a general system for making differentially private queries over distributed user data. SplitX is typically two to three orders of magnitude more efficient in bandwidth, and from three to five orders of magnitude more efficient in computation than previous comparable systems (i.e., the systems with similar features and trust model). SplitX is resistant to answer pollution by malicious clients.
- It presents both security and performance evaluations of SplitX.
- It describes a complete implementation of SplitX, and reports the results from a realistic deployment across 416 clients distributed globally.

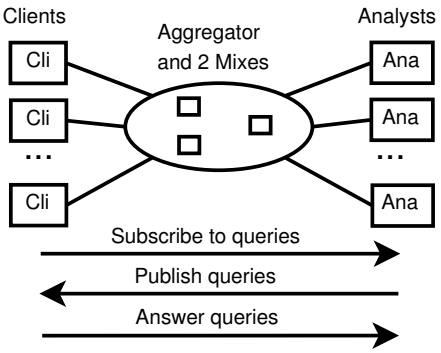

Figure 1: System overview. All interactions are anonymous and unlinkable. Answers are differentially private.

The rest of this paper is organized as follows. The next section gives an overview of SplitX, and states its goals and trust assumptions. Sections 3, 4, and 5 present the system design. We provide SplitX's privacy analysis in Section 6, and describe our implementation, performance evaluation and deployment in Section 7. Finally, we present related work and conclude in Sections 8 and 9, respectively.

2. SPLITX OVERVIEW

2.1 System Components

Figure 1 gives a high-level overview of SplitX, which consists of the following system components:

Analysts. An analyst is a server that formulates queries, publishes the queries to clients, either directly or indirectly via the aggregator, and receives the aggregate noisy results from the aggregator. There can be many analysts.

Clients. Clients store user data, and answer queries over that data. A client may be a user's own device, or some other device the user trusts with his or her data. Clients subscribe to queries from specific analysts. In some cases, a client may have a direct (non-anonymous) relationship with an analyst, and request queries directly from the analyst. One such example is web analytics, where the client could receive queries from the web server (i.e., analyst) when it accesses the web server, similar to [4]. In other cases, a client may have an indirect relationship with the analyst. An example is a software vendor that wishes to gather aggregate data on its user base. In this case, the client anonymously requests queries via the mixes and the aggregator. Client software can be bundled with existing software that requires private analytics, thus eliminating need for user incentives.

Aggregator and Mixes. The aggregator and two mixes interpose between analysts and clients. They work together to ensure that requests for queries and answers to queries are handled in a way that satisfies our privacy goals (§2.2).

The mixes receive XOR-encrypted answers from clients, blindly add additional XOR-encrypted random answers (as the differentially private noise [11, 12, 14]), shuffle the real and noisy answers, and pass them to the aggregator. The aggregator interfaces with analysts. It decrypts the answers received from the mixes, adds them up, and supplies the aggregate result to the appropriate analyst.

In addition, the mixes act as anonymizing proxies for query requests. Both mixes and the aggregator act as anonymizing proxies for query answers. Details are given in §4.

2.2 Privacy Goals

SplitX achieves three privacy properties: *anonymity*, *unlinkability*, and *differential privacy*. Anonymity means that no system component can associate user data with a user identifier. Unlinkability means that no system component can link any pair of requests or answers to the same user, even in the absence of an identifier (i.e., anonymously). Breaking the unlinkability property could allow system components to build quasi-identifiers which may in turn be de-anonymized with auxiliary information. Finally, all aggregate results in the system are produced under differential privacy guarantees, i.e., the produced aggregate results do not violate the privacy of any individual users.

2.3 Other Goals

The system should scale well. First, the system should be able to direct queries from a given analyst to only those clients which are associated with that analyst. Second, the system should scale at least linearly with 1) the number of clients, 2) the number of queries, 3) the size of queries, and 4) the size of answers. Third, the scaling factor should be small (e.g., efficient XOR operations versus expensive public-key operations). This both allows the system to handle millions of clients each answering thousands of queries, and accommodates low-capacity clients (e.g., mobile devices).

The system should tolerate churn among clients, since they may be frequently-disconnected devices like smartphones.

The system should be resistant to clients attempting to pollute results. In particular, a single answer should not be able to distort the aggregate result by more than a single count. While an ideal goal would be to limit any single device to a single answer per query, accomplishing this would amount to solving the Sybil problem, which is out of the scope of this paper. Rather, SplitX is able to detect duplicate answers per query that come from any given IP address.

2.4 Trust Assumptions

Clients are potentially malicious in that they wish to pollute aggregate results.

Analysts are potentially malicious in that they try to break the privacy properties described in §2.2, i.e., de-anonymize clients, link client requests or answers, or remove differentially private noise.

The *aggregator* and two *mixes* are honest but curious. They run the specified protocol faithfully, but may try to exploit additional information that can be learned in doing so. The aggregator does not operate fake clients. There is no collusion (i.e., information sharing) among the aggregator and the two mixes. In other words, at least two of the three entities refuse to collude.

While our honest-but-curious trust assumption is not new (see [4,9,20]), we wish nevertheless to emphasize that we believe that this assumption is appropriate in a realistic analytics scenario, for instance, where the aggregator pays mix operators for running mixes, and charges analysts. Here, the aggregator and the mixes would explicitly state non-collusion in their privacy statements, which are legally binding documents. Although the relationship between the aggregator and the mixes does create an opportunity for collusion, it would be difficult and risky for either party to leverage the individual user data obtained through that collusion. This is because the data analyst (e.g., an advertiser) would want to know the origin of the data, and an honest analyst would not only reject illicitly gathered data, but might also expose the aggregator or the mixes. As a result, we believe our trust assumption is reasonable in the real world.

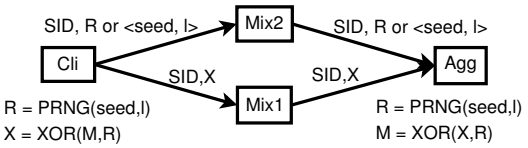

Figure 2: XOR-based encryption. Client Cli sends message M of length l to aggregator Agg via mixes Mix1 and Mix2.

Finally, we assume that crypto operations are operated correctly: messages cannot be eavesdropped, the aggregator and mixes cannot be impersonated, and changes in messages by a man-in-the-middle can be detected.

3. TECHNICAL BUILDING BLOCKS

Before we elaborate on the detailed design of SplitX in §4, this section first describes a few techniques that we will utilize throughout the whole design.

3.1 XOR-based Encryption

SplitX derives its performance from using the XOR operation as the basis of message encryption. XOR-based encryption has been previously proposed for anonymous communications [8]. In SplitX, it provides two key benefits:

- It is very efficient (relative to public-key operations).
- It allows mixes to blindly add noisy answers and shuffle them with real client answers (see §4.2).

This section describes the XOR-based encryption and its notation. Figure 2 illustrates the transmission of a message M of length l bits from client Cli to aggregator Agg via a pair of mixes Mix1 and Mix2. To do so, Cli generates a random string R, also of length l bits. In particular, R is generated with a pseudo-random number generator (PRNG) seeded with a cryptographically strong random number. Cli does an XOR operation with M and R to produce a new message X:

$$X = M \oplus R \tag{1}$$

Cli then sends X to Agg via Mix1, and sends either R or $\langle seed, l \rangle$, whichever is shorter, to Agg via Mix2. We call this operation *splitting*, and refer to the two XOR-encrypted messages as *split messages*. If Agg receives $\langle seed, l \rangle$, it can recreate R by seeding the same PRNG with the received seed. To recreate message M, Agg does an XOR operation with X and R:

$$M = X \oplus R \tag{2}$$

This operation is called *joining*. Because Agg may receive many split messages from Mix1 and Mix2, a pair of associated split messages (i.e., X, and R or $\langle seed, l \rangle$) must be identified as belonging to the same message. Therefore, Cli also generates a unique identifier per message called the split identifier (SID). The SID is itself a large random number, so that it is unique among all SIDs with high probability, during the time the message is being processed in the system.

Provided that the seed is cryptographically strong, each pair of the split messages appear random, and neither Mix1

nor Mix2 can decrypt the original message unless they collude (i.e., share information). To prevent an eavesdropper from decrypting messages, all direct communications between clients, mixes, and aggregator are encrypted with the standard transport layer security (TLS). To prevent spoofing, the mixes and aggregator are authenticated with public-key certificates.

3.1.1 Split Message Notation

The messages shown in Figure 2 can be fully notated as:

$$\begin{array}{l} Cli \xrightarrow{Mix1} Agg : SID, X \\ Cli \xrightarrow{Mix2} Agg : SID, R \text{ or } \langle seed, l \rangle \end{array} \quad (3)$$

For readability, however, we use the notation:

$$\begin{array}{l} Cli \xrightarrow{Mix1} Agg : \underline{M} \\ Cli \xrightarrow{Mix2} Agg : \underline{M} \end{array} \quad (4)$$

Or, equivalently:

$$Cli \xrightarrow[Mix2]{Mix1} Agg : \underline{M} \quad (5)$$

Here, the underline denotes the split of message M (either X, or R / $\langle seed, l \rangle$, it does not matter which). SID is omitted, and is understood to have been sent. Therefore, messages (4) or (5) may be read to mean "Cli sends split messages of M to Agg via Mix1 and Mix2".

More generally, Cli may send Agg a message with multiple fields, some of which are XOR-encrypted (i.e., split fields), and some of which are clear-text. This is denoted as:

$$Cli \xrightarrow[Mix2]{Mix1} Agg : \underline{F_{s1}, \cdots, F_{sn}}, F_{c1}, \cdots, F_{cm} \quad (6)$$

Here, $\underline{F_{s1}, \cdots, F_{sn}}$ are split fields, and $F_{c1}, \cdots, F_{cm}$ are clear-text fields.

3.2 Query Buckets

SplitX supports the SQL query language (and potentially other query languages). While queries formulated by analysts can be complex, the results of a query are expressed as counts within histogram *buckets*, where each bucket represents one possible answer value. The client's answer to a query is in the form of a '1' or a '0' per bucket, depending on whether or not the answer falls within that bucket.

There are two types of queries: numeric queries, and non-numeric (string) queries. For numeric queries, buckets are specified as numeric ranges. An example of numeric ranges is age. An analyst could learn the age distribution among males with an SQL query, such as "`SELECT age FROM splitx WHERE gender='male'`" by defining, for instance, 5 buckets as follows: 'age 0∼19', 'age 20∼39', 'age 40∼59', 'age 60∼79', and 'age≥80'. If a given male user is 30 years old, the client answers '1' for the second bucket, and '0' for all other buckets.

For non-numeric (string) queries, each answer bucket is specified by a regular expression. One such example is the websites visited. An analyst could learn the popularity of websites within users in the US with an SQL query, such as "`SELECT website FROM splitx WHERE location='US'`" with hundreds of thousands of buckets defined as, for instance, '*.google.com', '*.facebook.com', '*.yahoo.com', and so on. For each website such a user has visited, the client places a '1' in the corresponding bucket, and a '0' in others.

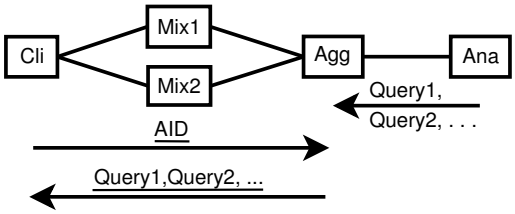

Figure 3: Query publish/subscribe. AID is analyst identifier. Underline denotes split message.

3.3 Differential Privacy

Differential privacy [11,12,14] is a privacy mechanism that protects user privacy by making it very hard to determine whether or not an individual user's record is in a queried database. A computation C achieves (ϵ, δ)-differential privacy [13] if, for any two data sets S_1 and S_2 that differ on at most one record, and for all outputs $O \subseteq Range(C)$:

$$\Pr[C(S_1) \in O] \leq \exp(\epsilon) \times \Pr[C(S_2) \in O] + \delta \quad (7)$$

That is, the probability that a computation generates a given output is almost independent of the presence of any individual record in the data set. There are two privacy parameters, ϵ and δ, in expression (7). They control the tradeoffs between the accuracy of a computation and the strength of its privacy guarantees. They also allow the amount of privacy leakage to be quantified and controlled [13].

Differential privacy is achieved by adding noise to the output of a computation. More specifically, the SplitX system generates some number of additional random answers as the differentially private noise, and randomly shuffles them with the real answers produced by clients. This is done independently for each query bucket. It is shown in [9] that adding n random answers achieves (ϵ, δ)-differential privacy:

$$n = \lfloor \frac{64 \ln(2c)}{\epsilon^2} \rfloor + 1 \quad (8)$$

Here, c is the number of clients answering the query, and ϵ is the privacy parameter which mainly controls the accuracy/privacy tradeoff [13]. In SplitX, the noise is added *blindly*, i.e., the system adds noise without knowing how much noise is added (see §4.2). The blind noise addition prevents the system from determining the noise-free aggregate result if, for instance, the noisy aggregate result were publicly released.

4. SPLITX SYSTEM DESIGN

With the technical building blocks in place, this section gives the detailed system design of SplitX.

4.1 Query Publish/Subscribe

Figure 3 illustrates how clients (Cli) anonymously request queries from analysts (Ana). Each analyst has an analyst identifier AID. Each analyst transmits its set of queries $Query_1, \cdots, Query_n$ to the aggregator (Agg). Each client requests the queries associated with a given analyst by transmitting the split AID via the mixes:

$$Cli \xrightarrow[Mix2]{Mix1} Agg : \underline{AID} \quad (9)$$

The aggregator joins (i.e., XOR-decrypts) the AID, and retrieves the AID's associated queries. Then, the aggregator

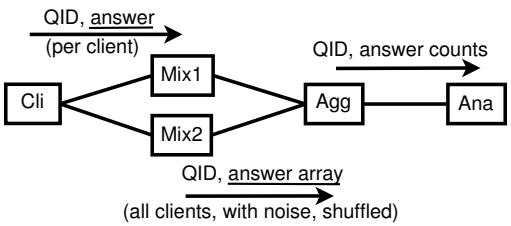

Figure 4: Simple but incomplete design for answering queries. QID is query identifier. The client sends its split answers each to one of the two mixes. Each mix adds split noisy answers, shuffles them with clients' split answers, and sends all split answers as an array to the aggregator. Aggregator reports the aggregate result to the appropriate analyst.

splits these queries, and transmits them to the client via the two mixes:

$$Agg \xrightarrow[Mix2]{Mix1} Cli : \underline{Query_1, \cdots, Query_n} \tag{10}$$

In SplitX, each query consists of the following fields:

$$Query := \langle QID, SQL, B_{1 \to b}, \epsilon, T_{end} \rangle \tag{11}$$

The query identifier QID uniquely identifies the query among all queries that the SplitX system may handle during the same time period. The QID may be composed of the analyst identifier AID concatenated with a value unique to the analyst.

SQL denotes the SQL query itself. $B_{1 \to b}$ denotes the b buckets which represent all possible answer values to the query. Examples of SQL queries and associated query buckets can be found in §3.2.

In addition, each query also contains a privacy parameter ϵ indicating how much differentially private noise should be added (see §3.3), as well as a query end time T_{end} indicating when answers to this query will no longer be accepted.

Note that the aggregator runs a sanity check over all the fields in the query. Most importantly, it ensures that the privacy parameter ϵ does not exceed the maximum allowable privacy level. It may also check to ensure that buckets do not overlap, and that the query end time T_{end} has not expired. Clients run a similar check.

4.2 Answering Queries

For descriptive clarity, this section starts with a simple but incomplete design, as shown in Figure 4. This design lacks some privacy properties. Subsequent sections (§4.2.1, §4.2.2, and §5) complete the design.

Step A1: Query Answering. Each client maintains its own data in a local database. How this database is obtained is outside the scope of this paper, though in our implementation we used database features of the Google Chrome browser. When a client receives a query from the analyst (see §4.1), it queries its local database and generates an answer. The client then maps the answer into b buckets, resulting in a '1' or a '0' per bucket, depending on whether or not the answer falls within that bucket. This answer is efficiently encoded as a bit-vector, with one bit per bucket, in the order that the buckets were received in the query.

$$Answer := \langle B_1, B_2, \cdots, B_b \rangle \tag{12}$$

The client splits the answer, and sends the split answers to the two mixes, respectively:

$$\begin{aligned} Cli \to Mix1 : QID, \underline{Answer} \\ Cli \to Mix2 : QID, \underline{Answer} \end{aligned} \tag{13}$$

Step A2: Answer Collection. Upon reception of a client's split answer, the mix stores the answer locally. If the mix has not seen the query identifier QID before, the mix requests the associated query from the aggregator. As a result, the mix knows the query's privacy parameter ϵ and the query end time T_{end}.[1]

Step A3: Mix Synchronization. After T_{end} expires for a given query identified by QID, both mixes must add noisy answers to the set of answers so far received, and randomly shuffle all answers (both noisy and real). Note that the mixes are working on split answers, not clear-text answers. Therefore, for each split noisy answer added by one mix, the other mix must add a corresponding split noisy answer. Later the aggregator will join each bit from each split answer. For this to work properly, the two mixes must be perfectly synchronized during this operation. This synchronization requires that:

1. The mixes agree on the exact set of real client answers they use[2], and
2. The mixes agree on a seed value of the pseudo-random number generator (PRNG), used for the synchronized operations in noise addition (step A4) and answer shuffling (step A5).

To achieve the first requirement, one of the two mixes takes the role of master. The master mix sends its complete list L_1 of all received SIDs (i.e., split identifiers of the received split answers) to the slave mix. The slave mix compares L_1 with its own set of SIDs, and removes all those that are not in L_1. The slave mix then returns to the master the list L_2 of split identifiers that the master has but the slave does not have. The master will remove these from its own SIDs. At the end of this exchange, the two mixes agree on the exact set of SIDs.

To achieve the second requirement, the master mix generates a cryptographically strong random seed s, and transmits it to the slave mix during the mix synchronization. Then, both master and slave mixes can initialize a PRNG with the synchronized seed s. We call this synchronized PRNG as $PRNG_{sync}$. In addition, for generating random noisy answers in step A4, each mix initializes another PRNG with a locally generated (unsynchronized) random seed, called $PRNG_{local}$.

Step A4: Noise Addition. According to equation (8) in §3.3, each mix determines n, i.e., the number of noisy answers that need to be added to achieve the required differential privacy. Using $PRNG_{local}$, each mix generates n split answers, with a uniformly random '1' or '0' for each bucket in each split answer. Then, using $PRNG_{sync}$, each mix generates the same set of n split identifiers SIDs, and assigns each of them to one of these n locally generated split noisy answers.

[1]With this simple design, the mixes can at this point associate QID with clients, and thus, learn which analysts each client is associated with. This is a violation of our privacy goals. We extend the design in §4.2.1 to solve this problem.

[2]It is possible that one mix receives a split answer while the other mix does not receive the pairing split answer.

	$B_1 B_2 B_3 .. B_b$	$B_1 B_2 B_3 .. B_b$
SID_a	$a_1\ a_2\ a_3 .. a_b$	$c_1\ b_2\ c_3 ..\ e_b$
SID_b	$b_1\ b_2\ b_3 .. b_b$	$a_1\ c_2\ e_3 ..\ a_b$
SID_c	$c_1\ c_2\ c_3 .. c_b$	$d_1\ z_2\ d_3 ..\ z_b$
SID_d	$d_1\ d_2\ d_3 .. d_b$	$b_1\ e_2\ a_3 ..\ c_b$
SID_e	$e_1\ e_2\ e_3 .. e_b$	$z_1\ d_2\ z_3 ..\ b_b$
..	.. ..	.. ..
SID_z	$z_1\ z_2\ z_3 .. z_b$	$e_1\ a_2\ b_3 ..\ d_b$

Figure 5: Answer shuffling. The left two-dimensional array is before answer shuffling, and the right two-dimensional array is after answer shuffling.

Note that later the aggregator will join each pair of these split noisy answers to obtain the non-split noisy answers. However, since neither mix knows the split noisy answers generated by the other mix, neither mix can know the resulting noisy answers at the aggregator. In this way, the mixes are able to add differentially private noise *blindly*.

Step A5: Answer Shuffling. At this point, the two mixes agree on the complete set of $c+n$ split answers, c from clients, and n from added noise. To shuffle the split answers, each mix creates a two-dimensional array of bucket values (see Figure 5). The rows of the array are split answers, in the order of lowest SID to highest SID. The columns of the array are buckets. Using the synchronized $PRNG_{sync}$, each mix goes through each column and randomly shuffles the values in the column. Note that, since both mixes use $PRNG_{sync}$ to shuffle, each bit position in their respective shuffled arrays corresponds to the same answer value.

Figure 5 illustrates this answer shuffling. The left array shows all the split answers in their original form. After the shuffling (i.e., in the right array), each column is still populated with the split answers for the corresponding bucket, but randomly shuffled. In doing so, the bucket values of each split answer have been completely de-correlated. This makes it prohibitively difficult to link a set of bucket values to a given client.

Finally, each mix transmits the shuffled split answer array to the aggregator. Since the two mixes effectively add n random noisy answers to the real client answers, this introduces the differentially private noise with a mean of $n/2$ to each individual bucket. For the final aggregate result adjustment, each mix also informs the aggregator of n:

$$\begin{array}{l} Mix1 \rightarrow Agg : QID, n, \underline{AnswerArray} \\ Mix2 \rightarrow Agg : QID, n, \underline{AnswerArray} \end{array} \quad (14)$$

Step A6: Noisy Answer Tabulation. Upon reception of the two split answer arrays, the aggregator generates the non-split answer array by joining each bit position in the two split answer arrays. The aggregator then sums together the values for each bucket, and subtracts $n/2$ to produce the final noisy count for how many clients fall within each bucket. Finally, the aggregator transmits the noisy counts to the appropriate analyst.

4.2.1 Anonymity and Unlinkability at the Mixes

The problem with the design as so far stated is that the mixes can associate QID with client in step A2, which violates our privacy goals. For instance, if there is an analyst per website, the mixes can learn which websites a client visits by monitoring which QID values are issued by which analysts. This section enhances the design to solve this problem.

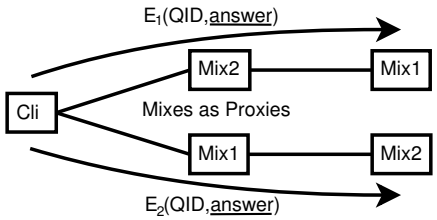

Figure 6: Mixes as proxies based on transport layer security. $E_1(M)$ and $E_2(M)$ denote the TLS encryption of message M destined to Mix1 and Mix2, respectively.

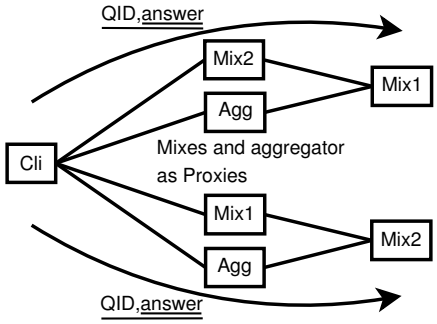

Figure 7: Mixes and aggregator as proxies based on double-splitting.

The specific goal here is to deliver the client's split answer (see expression 13) to the mix *anonymously* and *unlinkably*. Anonymous means that the mixes cannot identify which client sent a given split answer. Unlinkable means that the mixes cannot determine if any two split answers came from the same client or from different clients.

A straightforward solution would be to interpose proxies between clients and mixes, and to run transport layer encryption (i.e., TLS) between clients and mixes. Indeed, since we have two mixes, they can each operate as a proxy on behalf of the other (see Figure 6). Here, each client must establish a separate TLS session for *every* split answer (along with the associated QID); otherwise, the mix could use the TLS session to link an (anonymous) client to multiple split answers, and in turn to multiple QIDs. With auxiliary information, this set of QIDs could be used to de-anonymize a client especially when the client answers only a part of received queries.

While this approach works, we can exploit XOR encryption to make it much more efficient. This is because XOR encryption avoids the expensive per-answer TLS session establishment. Since XOR encryption requires a pair of proxies, we can additionally use the aggregator as one of the proxies in each pair (see Figure 7). Here, the client *double-splits* the message. In other words, it takes each of the original split messages (destined to Mix1 and Mix2 acting as mixes), and splits each again. The complete message contents, including the SID of the original split, are covered in the double split:

$$\begin{array}{l} Cli \xrightarrow[Agg]{Mix2} Mix1 : \underbrace{\underline{QID, \underline{Answer}}}_{\text{double-split answer}} \\ Cli \xrightarrow[Agg]{Mix1} Mix2 : \underbrace{\underline{QID, \underline{Answer}}}_{\text{double-split answer}} \end{array} \quad (15)$$

While there are four double-split messages in total, only one of them needs to be the full size of the client answer. That is because one of the original split messages contains $\langle SID, seed, l\rangle$ as described in §3.1, and the two splits of that message are likewise small. Note that double-splitting does not fully eliminate the need for TLS: the client needs to establish TLS sessions with proxies so that eavesdroppers cannot join messages. However, these per-proxy TLS sessions can be long-lived, and so session establishment is rare.

4.2.2 *Preventing Traffic Analysis*

In SplitX, there are opportunities for eavesdroppers and proxies (i.e., mixes and aggregator acting as proxies) to analyze encrypted traffic and deduce which queries are being answered by which clients. To prevent this, standard traffic analysis mitigation techniques are used as follows.

Fixed-Size Messages. All transmitted messages are fixed at one of a set of exponentially growing message sizes. Very long queries (e.g., those with millions of buckets) are partitioned into multiple smaller queries, each with the same SQL but different bucket definitions. Small queries from the same analyst are bundled into a single message, and their answers are likewise bundled together. This not only makes traffic analysis harder, but improves the system efficiency. All messages are padded to the next bigger fixed size.

Delay and Reordering. Proxies randomly delay and reorder messages. Each message must be delayed long enough that it can be randomly reordered with some 100's of other messages of the same size.

5. POLLUTION ATTACK AND DEFENSE

In the design as described in §4, no system components can detect which queries are being answered by which clients. This leaves SplitX open to an *answer pollution* attack whereby a client simply answers the same query many times. Without the mechanism described in this section, such an attack would go unnoticed.

A simple defense is for the system proxies to limit a client to some fixed rate of answers. While this is simple, it is already a significant improvement over several previous systems [20, 26, 28], where even a *single* malicious client with a *single* answer can substantially distort the aggregate result.

Unfortunately, a client may legitimately answer queries for hundreds of analysts in a given query time period, for instance, one for each website the user visited, and each application the user installed. This section presents a mechanism that detects duplicate answers received from any given IP address, removes those answers, and tags IP addresses that associate with duplicate answers.

5.1 Duplicate Detection Mechanism

Figure 8 shows the system topology with the duplicate detection mechanism. Compared to Figure 7, an additional split is added between the client and the aggregator-as-proxy. The purpose of this split is to create a setup whereby three system components can cooperate to *blindly* perform duplicate detection. That is, none of the three system components can de-anonymize or link client data. Specifically, as shown in Figure 8, the three components are the ones explicitly labeled "Proxy" (i.e., Mix1), "Mix" (i.e., Mix2), and "Agg" which lies between them. Hereafter, we will use Proxy, Mix, and Agg to represent these three specific components.

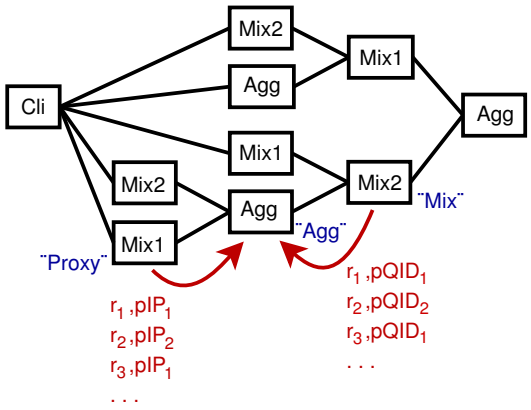

Figure 8: System topology based on triple splitting, with messages used for duplicate answer detection. r_i is a unique answer identifier. pIP_j is a pseudonym for the client IP address. pQID_k is a pseudonym for the query identifier.

The purpose behind this setup is to create 1) a component (i.e., Proxy) that knows the client IP address but not the query identifier QID, 2) a component (i.e., Mix) that knows the QID but not the client IP address, and 3) a component (i.e., Agg) that knows neither. We now describe the duplicate detection mechanism.

The Proxy and Mix establish a TLS-encrypted channel between them. This channel allows the Proxy to send encrypted message fields to the Mix without the intervening Agg being able to read them. The Proxy assigns a pseudonym pIP to each client IP address. The Mix assigns a pseudonym pQID to each QID.

Step D1: The Proxy assigns a unique random number r to each (triple-split) answer message received from client. The Proxy encrypts r, and sends it along with the answer message to the Agg. After joining the two triple-split messages from the Proxy and Mix2, the Agg attaches the encrypted r to the joined (double-split) message, and sends it to the Mix.

Step D1': After a random delay, the Proxy also sends the $\langle r, \text{pIP}\rangle$ tuple to the Agg. This tuple informs the Agg that r is associated with an answer received from a client with the pseudonym pIP.

Step D2: Upon reception of the (double-split) message from the Agg in step D1, the Mix decrypts r. After joining this message with another (double-split) message received from Mix1, the Mix knows the QID (see §4.2.1), and therefore its associated pseudonym pQID. As a result, the Mix can generate a tuple $\langle r, \text{pQID}\rangle$, indicating that r is associated with an answer to a query with the pseudonym pQID.

Step D3: After a given query expires, but before the mix synchronization process (i.e., step A3 in §4.2), the Mix transmits all $\langle r, \text{pQID}\rangle$ tuples to the Agg. The Agg matches tuples with the same r to create a list of $\langle r, \text{pIP}, \text{pQID}\rangle$ tuples. Any such tuples with the same pIP and pQID represent a duplicate answer. That is, a client answered the same query more than once.

Step D4: The Agg returns the r values of all duplicate answers to both the Mix and the Proxy. The Mix removes these from its list of split answers, and then continues with the mix synchronization process. The Mix also uses the information to detect whether an analyst may be under a

pollution attack (i.e., an unusually high percentage of duplicate answers). The Proxy likewise uses the information to tag suspicious clients that are sending too many duplicates.

5.2 Mitigating Linkability

While the duplicate detection mechanism so far described provides anonymity, without additional measures it suffers from linkability. In the mechanism, the Agg can link a pIP with a pQID. If the Agg can deduce the QID associated with each pQID, then it can link the QID to (anonymous) clients. Without the following mechanisms, the Agg could derive QID from pQID by examining the query end time and the number of answers for each pQID.

Synchronized Query Epochs. Time is divided into epochs. Queries' start and end times are synchronized to those epochs, i.e., queries are synchronized with other queries so that their start and end times are the same. This makes it hard for the Agg to deduce the QID based on the query's start and end times.

Fixed-Number Query Answers. In step D1, the Proxy routinely generates unique random numbers r, and then uses them to generate some number of extra $\langle r$, pIP$\rangle$ tuples. The Proxy sends these extra tuples to the Agg, and sends the encrypted r's to the Mix. The Mix decrypts those encrypted r's. Based on those r's, the Mix transmits some number of fake $\langle r$, pQID$\rangle$ tuples to the Agg, such that the total number of tuples for pQID is fixed at one of a set of exponentially growing sizes. In doing so, many queries associate with the same number of $\langle r$, pIP, pQID$\rangle$ tuples. As a result, the Agg cannot derive the QID from the pQID based on the number of tuples associated with pQID.

5.3 Discussion

The duplicate detection mechanism removes all duplicate answers received from any given IP address. This may have the effect of biasing the query result, for instance, against business users where many users may operate behind a single NAT box. To mitigate this bias, the Agg in step D4 can choose to report the r values of all but k randomly selected duplicate answers. There is a tradeoff of choosing k. If k is too high, then a malicious client could answer the same query k times without detection; on the other hand, if k is too low, this may introduce bias. The best strategy depends on the application domain. Note also that, in a mobile environment, each client may have an authenticated unique identity, e.g., the IMEI number. We can use this identity (rather than the IP address) as the client identifier to solve the problem associated with NAT, as well as the problem with clients changing IP addresses.

Recall from the trust assumptions in §2.4 that the aggregator is honest but curious. However, if it is *malicious*, the aggregator could operate fake clients in an attempt to link clients to analysts by exhibiting a unique signature of answers for a known set of queries. For instance, a fake client operated by the aggregator could generate 100 duplicate answers for QID_1, 200 duplicate answers for QID_2, and so on. By looking for this signature in $\langle r$, pIP, pQID$\rangle$ tuples, the aggregator could reverse-engineer the "pQID↔QID" mapping, and in turn link (anonymous) clients to analysts. One possible approach against the malicious aggregator could be to randomly remove some client answers and add additional fake answers in order to break the signature. The effectiveness of such an approach is a topic for future study.

6. PRIVACY ANALYSIS

This section analyzes the privacy properties for all SplitX system components.

6.1 Analyst

Analysts are considered potentially malicious, and may try to violate individual users' privacy by learning their real answers. In SplitX, the analyst interacts with clients either directly, or indirectly via the aggregator (see §2.1). Even in the case of a direct interaction, only the queries are sent from the analyst to clients; the (split) answers are *always* sent through the mixes and aggregator; thus, the analyst receives only aggregate results with differentially private noise.

The analyst may try to manipulate its query's fields, i.e., $\langle QID, SQL, B_{1\to b}, \epsilon, T_{end}\rangle$ in expression (11). However, no setting of these fields prevents the mixes from adding differentially private noise to the aggregate result. Of course, the analyst may set a large ϵ in an attempt to lower down the differentially private noise being added to the aggregate result. However, both the aggregator and the clients will run a sanity check to discard those queries (see §4.1).

6.2 Aggregator

The aggregator sees the analyst identifier in the query request (see §4.1). However, it does not know which client made that request, because the mixes interpose between the clients and the aggregator (see Figure 3).

The aggregator (not acting as proxy) receives split answers for a query from the two mixes after the query end time. These split answers contain differentially private noise, so that the aggregate result produced from these answers does not violate the privacy of any individual client. Furthermore, the bucket values in these received split answers are randomly shuffled at the mixes (step A5 in §4.2). This shuffling prevents the aggregator from identifying a client based on a set of bucket values.

6.3 Mix

The mix (not acting as proxy) can join a pair of double-split answers to recreate a split answer which includes the query identifier. However, it cannot determine which client sent this split answer. This is because the double-split answers are interposed by the other two system components acting as proxy.

Each mix adds split noisy answers. However, neither mix knows the split noisy answers added by the other mix, such that they do not know how much differentially private noise is added to the aggregate result (step A4 in §4.2). What each mix knows is only that the noise added is enough to achieve the required differential privacy. As a result, even if the noisy aggregate result were publicly released, neither mix can subtract the noise to produce the noise-free result.

6.4 Proxy

The aggregator and mixes act as proxies for each other, providing clients with anonymity. Furthermore, each proxy only sees one out of a pair of (double- or triple-) split messages. The proxy cannot link any two of these split messages. This is because each pair of split messages has a unique split identifier, and it is unlinkable to other pairs, even to those that are sent by the same client. This provides the unlinkability property for SplitX. Any traffic analysis attempts by the proxy will be thwarted by the "fixed-size messages" and

Table 1: Microbenchmarks (# query buckets / second). The public-key crypto schemes use a 1024-bit key.

	Splitting / Encryption			Joining / Decryption		
	Chrome (client)	Smartphone (client)	Server	Chrome (client)	Smartphone (client)	Server
SplitX	51,336,008	1,565,240	139,028,952	91,246,848	6,194,472	445,571,096
PDDP [9]	25,989	823	20,569	—	—	6,097
Akkus et al. [4]	1,441	57	9,814	—	—	668

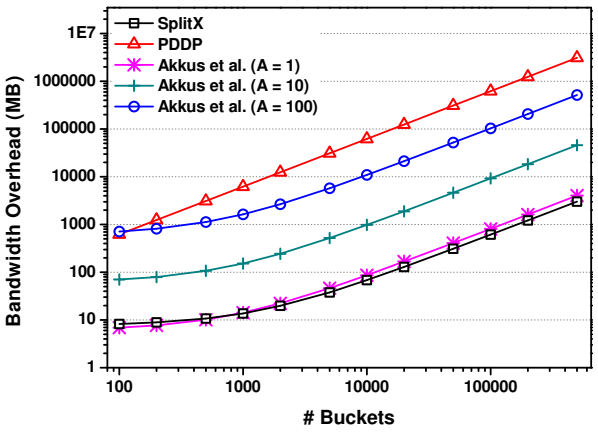

Figure 9: Bandwidth overhead at the aggregator. For SplitX, the seed value and split identifier are 128-bit and 64-bit long; the privacy parameter ϵ=1.0. For PDDP and Akkus et al.'s systems, the public key length is 1024-bit, and the privacy parameter ϵ=1.0.

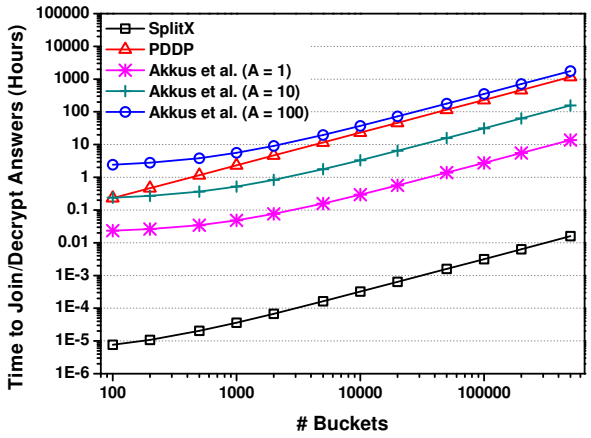

Figure 10: Computational overhead at the aggregator. For SplitX, the seed value and split identifier are 128-bit and 64-bit long; the privacy parameter ϵ=1.0. For PDDP and Akkus et al.'s systems, the public key length is 1024-bit, and the privacy parameter ϵ=1.0.

the "delay and reordering" mechanisms (see §4.2.2). Note that as a general rule, such chaff is never perfect, and over time the traffic analysis may link some clients to some analysts. In the extreme, one could deploy SplitX such that the proxies are not re-purposed mixes and aggregator, but rather completely separate entities. Such an approach would require as many as nine distinct entities (see Figure 8).

6.5 Client

Clients are considered potentially malicious with a goal of polluting the aggregate result. A malicious client may send duplicate answers for the same query, but these answers will be detected and discarded (see §5.1). Consequently, to bypass detection and pollute the aggregate result, an attacker must use a botnet. This case is not different from the situation today where, for instance, an attacker could build a botnet to produce false web visits in an attempt to distort the result produced by a service like Google Analytics.

A malicious client may also send unpaired split messages, or send only one split message but withhold the pairing split message to consume resources. These unpaired messages can easily be discarded after a joining operation or a timeout.

7. IMPLEMENTATION & EVALUATION

7.1 Implementation

We implemented the SplitX system. The client is a Google Chrome extension consisting of 1.7K lines of JavaScript code. The extension captures webpages browsed, searches made and extensions installed, and stores them into a local database with timestamps. In principle, the extension can be extended to capture any user activity within the browser.

All other system components were implemented in Java. In total, they consist of 5K lines of code. The interfaces between all system components were defined by 116 lines of code in the Apache Thrift interface description language [1], producing 19K and 5K lines of Java and JavaScript code, respectively. Apache Thrift enables SplitX to seamlessly support potential clients implemented in various languages.

7.2 Microbenchmarks

A major performance bottleneck in private analytics systems is the overhead of crypto operations. We compare SplitX with two recent systems, PDDP [9] and Akkus et al. [4], because they both are resistant to answer pollution and they both add noise in the infrastructure rather than at the client, making them most comparable to SplitX.

The crypto primitives of SplitX, PDDP, and Akkus et al. are splitting/joining operations, Goldwasser-Micali crypto operations [17, 18], and RSA operations, respectively. To evaluate the performance of these operations, we conducted microbenchmarks on a desktop computer with Intel dual core 3GHz running Linux 3.2.24, as well as on a smartphone with a 1GHz CPU running Android 2.3. In particular, we measure the number of query buckets that the three systems can process per second.

Table 3: Browsing activity of clients on a given day. Noisy counts are given for each bucket.

	Date	0	1∼10	11∼20	21∼50	51∼100	101∼200	201∼500	501∼1000	>1000	Total
#	Jan 27	123	47	27	43	8	0	0	1	1	250
searches	Jan 28	102	51	43	44	12	8	-2	2	-1	259
made	Jan 29	80	82	39	33	19	3	0	1	-1	256
#	Jan 27	101	3	9	10	21	20	56	21	8	249
webpages	Jan 28	76	9	6	20	14	32	60	27	11	255
visited	Jan 29	45	13	7	14	24	42	61	43	13	262
# unique	Jan 27	98	31	26	63	28	3	-1	-3	3	248
websites	Jan 28	77	32	39	70	28	5	-1	-1	2	251
visited	Jan 29	45	49	43	83	37	6	1	0	1	265

Table 2: Queries used in our deployment.

Query	# buckets	Type
Extensions/Apps installed	46.5K	String
Websites visited	400K	String
Product keywords used for search	400K	String
# searches made (daily)	9	Numeric
# webpages visited (daily)	9	Numeric
# unique websites visited (daily)	9	Numeric

The client microbenchmark is written in JavaScript. We measure two types of clients: the Google Chrome browser on the Linux machine and the WebKit browser on the smartphone. Table 1 shows, not surprisingly, that encryption at the client in SplitX is three or more orders of magnitude faster than that in PDDP and Akkus et al. SplitX is also extremely efficient for decryption at the client. PDDP and Akkus et al.'s systems are not designed to provide anonymity and unlinkability for query requests; thus, they do not need decryption at the client.

The server microbenchmark is written in Java and run on the Linux machine. Table 1 shows, again not surprisingly, that SplitX's crypto operations at the server are four or more orders of magnitude faster than the other systems.

7.3 System Overheads

To illustrate SplitX's performance gains and show the scalability of these systems, we compare SplitX with PDDP and Akkus et al. in terms of their bandwidth and computational overheads at the aggregator, which is the bottleneck of private analytics systems. We modeled a scenario, in which query answers are received from 50K clients for a query with a varying number of buckets from 100 to 500K. The bandwidth overhead at the aggregator is related to the number of messages received and the length of each received message. The computational overhead is the time required for the aggregator to obtain the aggregate result by joining or decrypting all received answer messages (including the client answers as well as the noisy answers).

Figures 9 and 10 show the aggregator's bandwidth and computational overheads, respectively. Akkus et al.'s system achieves its performance gains primarily by limiting the number of buckets that a client reports (i.e., A) to a small fixed value. In contrast, PDDP reports every bucket value like SplitX.

Regarding bandwidth, SplitX is similar to Akkus et al. with A=1, and one and two orders of magnitude more efficient than Akkus et al. with A=10 and A=100, respectively. Compared to PDDP, SplitX is at least two, and often three orders of magnitude more efficient.

Regarding computation, SplitX is more than three orders of magnitude more efficient than Akkus et al.'s system even with A=1. SplitX is over five orders of magnitude more efficient than PDDP.

Table 4: Top 15 most visited websites and most used extensions/apps by the clients.

Website	Noisy count of clients	Type
www.google.com	148	Website
www.facebook.com	144	Website
www.mturk.com	137	Website
www.youtube.com	127	Website
www.amazon.com	118	Website
www.surveymonkey.com	61	Website
www.imdb.com	52	Website
www.linkedin.com	50	Website
www.google.co.in	43	Website
www.ebay.com	40	Website
www.ehow.com	39	Website
www.reddit.com	34	Website
www.huffingtonpost.com	34	Website
www.yelp.com	33	Website
www.thefreedictionary.com	29	Website
Gmail	158	App
Youtube	148	App
Google Drive	81	App
AdBlock	39	Extension
Angry Birds	26	App
Turkopticon	25	Extension
Adblock Plus	20	Extension
Google Chrome to Phone	16	Extension
Reddit Enhancement Suite	14	Extension
IDM Integration	14	Extension
Google Calendar	14	App
LastPass	13	Extension
Poppit	12	App
Adblock for Youtube	11	Extension
Entanglement	11	App

7.4 Deployment

Before we deployed SplitX 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 doing so, we could verify the correctness of SplitX. To further demonstrate the feasibility, we then deployed SplitX across 416 unique clients via friends and Amazon Mechanical Turk (AMT). Our experiment lasted during January 21-31, 2013. There were between 225 and 313 active clients each day. Table 2 lists the queries we issued throughout our experiment. For *string* queries, we obtained their respective bucket values from 1) Chrome Web Store, 2) Alexa, and 3) a set of product descriptions gathered from the Amazon, shopping.com, and

Google product APIs. For our queries, we used the privacy parameter ϵ=5, which for our client population generates around 16 noisy answers per query, and produces a normal distribution with the standard deviation σ=2, such that the noise in each bucket is within 2, 4, and 6 with probabilities of 68%, 95%, and 99.7%, respectively [9]. Note that in a commercial analytics setting with far more clients, a smaller ϵ would be used, leading to more noise being added. The increased noise would give clients better privacy without affecting the accuracy of the aggregate result, because it would be relatively small compared to the number of clients.

Table 3 shows the clients' browsing activity. Many clients were fairly active, having made up to 50 searches, browsed more than 50 webpages, and visited up to 100 unique websites on a day. Table 4, interestingly, reflects our AMT-based user population. Besides popular websites and apps one would normally expect, mturk.com has the third highest count among websites, and Turkopticon, an extension for feedback on AMT requesters, has the sixth highest count among extensions and apps.

7.5 Lessons Learned

While our experience with gathering user aggregates was positive, two lessons are worth mentioning. First, for the browsing activity query, we were not initially sure what ranges to set for the buckets. To avoid making multiple "end-to-end" (analyst to client) queries, we developed a protocol whereby the aggregator examines received bucket counts, merges buckets with low counts, and re-requests new noisy bucket counts from the mixes. This approach is both faster than re-querying clients, and avoids differential privacy leakage to the analyst. This protocol is not further discussed for lack of space.

Second, we found that querying for product-related search terms did not yield meaningful results. There are probably multiple reasons: the vocabulary was too big, the number of product-related searches was too small, and the vocabulary consisted of single terms rather than phrases. For queries like this, it would be more useful if the system could *discover* the appropriate vocabulary, rather than the analyst having to select it in advance. This is a topic for future research.

8. RELATED WORK

There is a massive literature on database privacy that assumes a trusted database, including [11,22,31] (see [16] for a survey). SplitX, by contrast, prevents any single system component (except for the user's own device) from viewing individual user data.

A number of systems utilize the XOR operation [8,29,33], or matrix multiplication [21] to achieve low-cost anonymization without public-key operations. These systems, however, support applications other than distributed differential privacy, and cannot be directly used.

Other systems target gathering data from distributed users in an anonymous and privacy-preserving way, but without using differential privacy. Applebaum et al. [5] uses a two-server architecture, in which one server provides anonymity to participants and obliviously blinds keys, and the other server aggregates blinded keys' values. The goal of preventing participants from learning other participants' keys creates a need for strong crypto operations, causing large overhead to each participant and preventing scalability in our setting. Anonygator [25] assumes that the content of the data collected is not going to leak the privacy of users. Similarly, P3 [23] requires an algorithm to determine which data would be safe to contribute a priori. Both of these systems also assume that the users are using an anonymity network to hide the source identity. In contrast, SplitX provides users with differential privacy guarantees that are independent of the content of messages.

A number of prior systems provide differential privacy in distributed settings. Some of these systems rely on complex crypto operations at the clients, causing high overheads [13]. To reduce this complexity, others deploy servers to distribute keys [26, 28]. The key distribution, however, suffers from client churn, which is inevitable in large-scale environments, rendering these systems unsuitable for our purposes. To deal with client churn, and still provide differential privacy in such environments, Hardt et al. [20] utilize two honest-but-curious servers, which cooperatively obtain the noisy aggregate result after the clients add noise. Unfortunately, these systems [20, 26, 28] allow a single answer by a single malicious client to pollute the query result substantially, whereas SplitX defeats this attack via duplicate detection.

P4P [10] prevents malicious clients from polluting the result with relatively efficient zero-knowledge proofs. However, this scheme is still computationally too costly to be practical in a commercial setting. In a more efficient way, PDDP [9] and Akkus et al. [4] prevent such malicious clients via buckets, similar to SplitX. In PDDP, one honest-but-curious proxy adds differentially private noise *blindly*. However, the proxy always knows which clients answer which analysts' queries, breaking unlinkability. Letting *all* clients answer *all* queries solves this problem, but this approach becomes unscalable and impractical very quickly. Akkus et al. [4] overcome this problem by utilizing the publisher (i.e., analyst for web analytics) as the proxy. Both systems, however, still require public-key crypto operations; thus, they are not nearly as scalable as SplitX.

Some systems store user data in an encrypted form and perform queries over that data [24, 27, 30]. These systems have different goals than SplitX. In particular, they produce *exact* results, allowing a malicious analyst to perform queries revealing individual user data. By contrast, SplitX produces differentially private results.

9. CONCLUSION AND FUTURE WORK

We designed and built SplitX, a system for distributed differential privacy, that is typically two to three orders of magnitude more efficient in bandwidth, and from three to five orders of magnitude more efficient in computation than previous comparable systems. We believe that this brings us one step closer to the goal of a practical and widely deployed private analytics system. SplitX assumes an honest-but-curious but nevertheless realistic trust model, and can scalably accommodate a large number of analysts, each working with different client segments. However, there are still significant research problems that need to be solved.

First of all, we would like to better explore the space of defending against a malicious aggregator. We should explore methods to make it harder for the aggregator to link clients and analysts, even under the assumption that the malicious aggregator can operate fake clients (see §5.3). We should also better understand the cost and effectiveness of these attacks and defenses, and analyze their security properties.

Currently, we cannot use SplitX in cases where the ana-

lysts do not know in advance the strings they wish to use as bucket values in string queries (see §7.5). A system that could automatically discover and report string values while still providing differential privacy guarantees would be valuable. This system could be used, for instance, in a search setting where the analyst cannot predict in advance what users may search for.

Finally, it is common in analytics systems that user data is constantly changing. As a result, it is necessary that analysts repeat the same query, e.g., daily or weekly. We would like to develop mechanisms that, if possible, minimize the theoretical privacy loss as defined by differential privacy, and if not possible, at least practically minimize the real risk of privacy loss. Two simple examples of such mechanisms would be to have clients refrain from answering a repeated query if the answer has not been changed, and have the system refrain from reporting results if too few clients answer.

10. ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers and our shepherd, Nina Taft, for their insightful comments. We also thank Sebastian Probst Eide, Alexey Reznichenko, and Nan Zheng for their valuable feedback on drafts of this work. We acknowledge Saikat Guha who suggested the bandwidth optimization scheme. Finally, we would like to thank the volunteers for anonymously helping us exercise our system.

11. REFERENCES

[1] Apache Thrift. `http://thrift.apache.org/`.

[2] Directive 2009/136/EC of the European Parliament and of the Council. `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2009:337:0011:0036:en:PDF`.

[3] Web Tracking Protection. `http://www.w3.org/Submission/web-tracking-protection/`.

[4] I. E. Akkus, R. Chen, M. Hardt, P. Francis, and J. Gehrke. Non-tracking web analytics. In *CCS*, 2012.

[5] B. Applebaum, H. Ringberg, M. J. Freedman, M. Caesar, and J. Rexford. Collaborative, Privacy-Preserving Data Aggregation at Scale. In *Privacy Enhancing Technologies*, 2010.

[6] M. Backes, A. Kate, M. Maffei, and K. Pecina. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *IEEE Symposium on Security and Privacy*, 2012.

[7] C. Castelluccia and A. Narayanan. Privacy considerations of online behavioural tracking. In *European Network and Information Security Agency (ENISA)*, 2012.

[8] D. Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *J. Cryptology*, 1(1):65–75, 1988.

[9] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards Statistical Queries over Distributed Private User Data. In *NSDI*, 2012.

[10] Y. Duan, J. Canny, and J. Z. Zhan. P4P: Practical Large-Scale Privacy-Preserving Distributed Computation Robust against Malicious Users. In *USENIX Security Symposium*, 2010.

[11] C. Dwork. Differential Privacy. In *ICALP*, 2006.

[12] C. Dwork. Differential Privacy: A Survey of Results. In *TAMC*, 2008.

[13] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *EUROCRYPT*, 2006.

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

[15] J. Freudiger, N. Vratonjic, and J.-P. Hubaux. Towards Privacy-Friendly Online Advertising. In *W2SP*, 2009.

[16] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Comput. Surv.*, 42(4), 2010.

[17] S. Goldwasser and S. Micali. Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information. In *STOC*, 1982.

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

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

[20] M. Hardt and S. Nath. Privacy-aware personalization for mobile advertising. In *CCS*, 2012.

[21] S. Katti, J. Cohen, and D. Katabi. Information Slicing: Anonymity Using Unreliable Overlays. In *NSDI*, 2007.

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

[23] A. Nandi, A. Aghasaryan, and M. Bouzid. P3: A Privacy Preserving Personalization Middleware for Recommendation-based Services. In *HotPETS*, 2011.

[24] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[25] K. P. N. Puttaswamy, R. Bhagwan, and V. N. Padmanabhan. Anonygator: Privacy and Integrity Preserving Data Aggregation. In *Middleware*, 2010.

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

[27] E. Shi, J. Bethencourt, H. T.-H. Chan, D. X. Song, and A. Perrig. Multi-Dimensional Range Query over Encrypted Data. In *IEEE Symposium on Security and Privacy*, 2007.

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

[29] E. G. Sirer, S. Goel, M. Robson, and D. Engin. Eluding carnivores: file sharing with strong anonymity. In *ACM SIGOPS European Workshop*, 2004.

[30] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *IEEE Symposium on Security and Privacy*, 2000.

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

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

[33] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in Numbers: Making Strong Anonymity Scale. In *OSDI*, 2012.