Diffix: High-Utility Database Anonymization

Paul Francis[1], Sebastian Probst Eide[2] and Reinhard Munz[1]

[1] Max Planck Institute for Software Systems,
Kaiserslautern and Saarbrücken, Germany
{francis,munz}@mpi-sws.org
[2] Aircloak GmbH, Kaiserslautern, Germany
sebastian@aircloak.com

Abstract. In spite of the tremendous privacy and liability benefits of anonymization, most shared data today is only pseudonymized. The reason is simple: there haven't been any anonymization technologies that are general purpose, easy to use, and preserve data quality. This paper presents the design of Diffix, a new approach to database anonymization that promises to break new ground in the utility/privacy trade-off. Diffix acts as an SQL proxy between the analyst and an unmodified live database. Diffix adds a minimal amount of noise to answers—Gaussian with a standard deviation of only two for counting queries—and places no limit on the number of queries an analyst may make. Diffix works with any type of data and configuration is simple and data-independent: the administrator does not need to consider the identifiability or sensitivity of the data itself. This paper presents a high-level but complete description of Diffix. It motivates the design through examples of attacks and defenses, and provides some evidence for how Diffix can provide strong anonymity with such low noise levels.

Keywords: Privacy, Anonymity, Analytics, Database

1 Introduction

The General Data Protection Regulation (GDPR) makes a distinction between *anonymization* and *pseudonymization*. Anonymized data "are outside the remit of data protection" [10]: none of the provisions in the GDPR for protecting personal data need apply. When data is anonymized, individuals cannot be "singled out", at least not without undue cost and effort. Pseudonymization — replacing Personally Identifying Information (PII) with irreversible identifiers — still allows for identifying individuals, typically through the use of external knowledge about those individuals [1]. The GDPR considers pseudonymized data to be personal data that must be protected, thus limiting its legal use.

The GDPR strongly encourages the use of anonymization, and wants to make it clear that pseudonymization is not enough. The problem is that anonymization renders data useless for most analytic needs. At best, substantial expertise and effort is required to anonymize data, and even then it is extremely difficult to

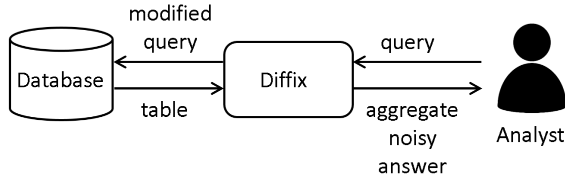

Fig. 1. Diffix acts as an SQL proxy to an unmodified database

determine whether data has been adequately anonymized or not[3]. Organizations that wish to share data are caught in a no-win situation: they can't share the data either because anonymization destroys the data, or because they are prevented from doing so because of concerns over legal liability.

This paper presents Diffix, a new approach to anonymization that substantially increases the utility of anonymized data while at the same time requiring no specialized expertise to configure. As Figure 1 illustrates, Diffix operates as an SQL proxy between the analyst and an unmodified database (which does not have to be a database with an SQL interface). Analysts submit standard SQL queries to Diffix, which in turn queries the database and computes a noisy answer that it returns to the analyst. The amount of noise is minimal (Section 7): typically Gaussian with a standard deviation of two for counting queries (those that count distinct users). Analysts may make an unlimited number of queries to the database. There is no restriction on the data types in the database, including for instance free text and event sequences.

Sections 2 through 4 describe the problem, state assumptions, and give related work. Sections 5 and 6 describe Diffix. Section 7 presents evidence to justify Diffix's low noise levels. Finally, Section 8 states the limitations of this work, and outlines the future research agenda for Diffix.

2 Why is Anonymization so Difficult?

For many decades before the advent of computers, census bureaus had the mandate to release statistical data while protecting privacy. This was historically done by releasing aggregate statistics—numbers that pertain to at least K individuals. Computer technology opened the possibility of releasing far more statistics, indeed of letting the public query data directly to produce whatever statistics might be of interest. As early as 1972, however, it was understood that

[3] For example, a state-of-the-art anonymization tool is ARX [9]. Usage of this tool requires among many other things that the user is able to classify data as identifying, quasi-identifying, sensitive, and insensitive; can create masking-based, interval-based, or order-based generalization hierarchies; and can understand and configure privacy models such as δ-presence, l-diversity, t-closeness, δ-disclosure, k-Anonymity, k-Map, (ϵ,δ)-differential privacy, and risk-based privacy models for prosecutor, journalist and marketer risks.

simply withholding answers that pertain to fewer than K individuals does not adequately protect privacy [8].

The central problem is the *intersection attack*. By way of example, imagine a database that only returns answers that pertain to more than $K = 4$ individuals. Suppose that an analyst makes two queries, one for the number of people in the CS department, and one for the number of men in the CS department:

```
SELECT count(*) FROM table
WHERE dept = 'CS' AND
        gender = 'M'
```

```
SELECT count(*) FROM table
WHERE dept = 'CS'
```

Suppose that there are 34 people in the CS department, and 33 men. Since both of these numbers are greater than 4, the database returns the answers. The analyst can trivially conclude that there is one woman in the CS department even though the database would have refused to provide that answer directly. Armed with this knowledge, the analyst can then learn more about the woman. For instance, the analyst can query for the sum of salaries of all people in the CS department and the sum of salaries of all men in the CS department, and by taking the difference determine the salary of the woman:

```
SELECT sum(salary) FROM table
WHERE dept = 'CS' AND
        gender = 'M'
```

```
SELECT sum(salary) FROM table
WHERE dept = 'CS'
```

In 1979, it was shown that an analyst with no prior knowledge about the contents of a K-limiting database that gives exact answers can infer substantial information about individual users using the intersection attack [3]. It was also shown that one way to prevent the intersection attack is to distort answers unpredictably, for instance by randomly rounding user counts up or down to a value divisible by five [8, 7] or removing random rows from the set of matching rows [2].

The problem of course is that random noise can be eliminated through averaging: causing a given answer to be repeated and taking the average value. In [2], Denning et.al. make the following statement:

> Rounding by adding a zero-mean random value (noise) is insecure since the correct answer can be deduced by averaging a sufficient number of responses to the same query. Rounding by adding a pseudorandom value that depends on the data is preferable, because then a given query always returns the same response.

Contrary to their own advice, Denning et.al. proposed using a pseudo-random value that depends not on the data, but rather on the query syntax. In this way, repetitions of a given query produce the same random sequence and therefore the same noisy answer. The problem with this solution, which the authors recognized, is that the analyst can still average out the noise by generating multiple different queries that all produce the same result, for instance by adding non-affecting conditions like "age < 1000".

Since 2003 [4] there has been renewed interest from the CS research community in the idea of adding random noise to answers, especially in the context of differential privacy [5]. With the exception of Rappor [6], however, to our knowledge all differential privacy proposals are susceptible to the averaging attack[4]. This inevitably leads to large amounts of noise and/or restrictions on the number of queries an analyst can make. Arguably these shortcomings are responsible for the overall lack of practical uptake of differential privacy.

Rappor, which we understand to be deployed by Google, is a distributed system whereby queries are sent to clients, each of which holds purely local information nominally about one user. Noise is derived from the fact that clients may lie about their data (randomized response). Rappor allows repeated answers without additional loss of privacy through a memoization feature: clients return the same randomized response to the same query as long as the true answer hasn't changed. The randomized response approach however results in a substantial amount of noise—distinct counts may be off by several hundred.

3 Measure of Anonymity

This paper measures anonymity in terms of the confidence with which the analyst can single out a user; that is determine that a single user has one or more given attribute values. With an intersection attack, an analyst can single out a user with a set of exact answers. Therefore, from the point of view of Diffix, anonymity is defined in terms of the confidence with which an analyst can determine the exact number of users that have one or more given attribute values. For instance, in the context of the intersection attack example in Section 1, the analyst wants to determine the exact number of people in the CS department and the exact number of men in the CS department. We refer to each such determination as a *guess*. We consider Diffix's anonymity to be strong when:

- The confidence the analyst has in a guess is not substantially higher than a guess the analyst could have made purely with external knowledge, or
- The probability of a substantially higher-confidence guess is very low.

This definition of privacy is similar to that used by Denning [3] as well as by the European Union Article 29 Data Protection Working Party Opinion on Anonymization [1]. Both of these focus on the ability to single out a user. In the case of Denning; the ability to identify that a given user has (or doesn't have) a given attribute value. In the case of Article 29; by identifying a user's records in a database. Diffix differs from these in that it incorporates the analyst's confidence and the probability of obtaining that confidence.

---

[4] Despite the thousands of papers on anonymity we could only find two that try to add noise in a way that depends on the data [2, 6]. This is why the paper cites so little related work.

4 Assumptions and Terminology

Our system setup consists of an *analyst* that queries a *database* via Diffix (Figure 1). The database is conceptually a single table organized as rows and columns. The columns may be of any type, so long as there are equalities and inequalities defined for the type (e.g. *column* = *value* or *column* < *value*) returning TRUE or FALSE. The database holds "raw" data: no perturbation on the values in the database is required, and no columns need be removed, for instance those containing personally identifying information like names.

We refer to the entity whose privacy is being protected as the *user*. The user may well be a device like a smartphone or a vehicle or even an organization. We require that each database table with individual user data has a column containing user identifiers. This is typically nothing more than the Primary Key or Foreign Key in the relational database. By convention we call this column the *uid*. We assume that every distinct user has one and only one distinct `uid`. A user may of course have more than one row.

The database may change over time. However, to protect anonymity in the face of changes, all changes to the database must be timestamped.

Diffix must be configured to know 1) which column contains the timestamps, and 2) which column contains the uids. No other data-specific configuration is required. Critically, the system operator need not understand the semantics of any other columns.

The amount of noise proposed in this paper assumes no strong correlations between columns among groups of users. If such correlations exist, and there is a risk that an analyst knows of the correlation, then the amount of noise must be increased in proportion to the size of the correlated group.

An analyst may make an unlimited number of queries.

Analysts may have substantial knowledge about the contents of the database, including full knowledge of most columns and full or nearly full knowledge of most rows.

This paper does not address timing or other side-channel attacks.

5 Diffix: A New Approach to Anonymization

As stated earlier, Diffix is deployed as an SQL proxy sitting between the analyst and a live database (Figure 1). The analyst could be a human or an application. The database could have an SQL interface or some other interface. In principle, Diffix does not have to be SQL-based, but as we discuss later Diffix needs to understand certain semantics of the query, so the query language should at least be simple, or composed of a simple subset of a more complex language like SQL.

5.1 Query Syntax

For the purposes of this paper we assume a simple SQL structure as follows:

```
SELECT columns, stat_functions()
FROM table
WHERE condition AND condition [ AND ... ]
GROUP BY columns
```

Here *stat_functions()* refers to statistical functions executed by Diffix that add noise to answers. The current implementation has the statistical functions count, sum, average, standard deviation, min, max, and median, but in principle Diffix could compute a wide variety of statistical functions.

The syntax for a condition is limited to 'column operator constant', where the operator includes equalities '=' and '<>' (not equal), inequalities '>=' and '<'. Inequalities must define a *range*, with '>=' on the left and '<' on the right.

Our implementation in fact supports a richer though still not complete subset of SQL. Notably, it allows JOIN operations, sub-queries (nested SELECT statements), a variety of math, string, and date functions, and SQL aggregate functions with GROUP BY. Diffix also places some limitations on the SQL. With the exception of the IN statement, the implementation prevents union semantics: OR for instance is disallowed. The implementation also places limits on the granularity of inequalities (greater than or less than), described further in Section 5.5.

Ensuring that the analyst cannot bypass these and other limitations entails considerable complexity and is outside the scope of this paper.

5.2 Key Insights

Our work on Diffix essentially picks up where Denning left off in 1980: that is, to *add noise, but make the noise dependent on the data*. In this paper, we refer to this general idea as *sticky noise*. Because the intersection attack depends on the analyst controlling the set of users that comprise a query answer, a natural design choice would be to make the noise dependent on the set of users that comprise a given answer. For instance, we could derive the seed for the Pseudo-Random Number Generator (PRNG) that produces the noise from the distinct set of users. This would defeat a naive averaging attack because the same answer would always have the same noise.

There are, however, two classes of attack against which this simple notion of sticky noise does not defend. The underlying problem is that this simple sticky noise defends against single repeated answers, but does not defend against combinations of answers. One class of attack is averaging attacks that exploit groups of answers. The other class is difference attacks. These exploit the mere fact that a pair of answers differ. In the following we give one example of each.

Split Averaging Attack Consider the following attack, which we call the *split averaging attack*. The analyst produces the pair of queries shown in Figure 2.

The sum of the counts of the two queries gives the number of users in the CS department, plus noise (here assuming that there is one row per distinct `uid`). Now repeat the pair of queries, this time using `age=21` and `age<>21`. This produces

```
SELECT count(*) FROM table
WHERE dept = 'CS' AND
        age = 20
```

```
SELECT count(*) FROM table
WHERE dept = 'CS' AND
        age <> 20
```

Fig. 2. One pair of queries from a split averaging attack to learn exact count of people in the CS dept. Subsequent query pairs use `age=21, 22, ...`

the same sum, but with a different noise sample because each individual answer is different. The pairs can be repeated with `age=22, 23, 24,` and so on. With enough samples, the noise can be averaged away and a high-confidence exact count produced. Given exact counts, the analyst could then for instance carry out an intersection attack [3].

Difference Attack For this attack, suppose that an analyst happens to know that there is only a single woman in the CS department. Let's call her the victim. The analyst could form the two queries shown in Figure 3.

```
SELECT salary, count(*) FROM table
WHERE dept = 'CS' AND
        gender = 'M'
GROUP BY salary
```

```
SELECT salary, count(*) FROM table
WHERE dept = 'CS'
GROUP BY salary
```

Fig. 3. Difference attack to learn the woman's salary

Both queries produce a histogram of salary counts. The left query definitely excludes the victim from all answers. The right includes the victim only in the bucket that matches her salary. If there is any pair of answers whose values differ, regardless of the values of the reported noisy counts, then the analyst knows that the victim is included in the right-side bucket and therefore knows her salary.

We call this a *difference attack*, because analysts need only detect a difference in two answers — they don't care what the values are.

Key Insight: Noise Layers Looking at both attacks, we observe that the analyst must modify *conditions* (e.g. `dept='CS'` or `gender='M'`) to carry out the attack. Suppose that, rather than add a single noise value for a single answer, we additionally add a noise value *per condition*?

For the moment, let's assume that these per-condition noise values are sticky in that the same condition produces the same noise value. Looking at the split averaging attack of Figure 2, we see that each query would have three noise values: one based on the set of users, one based on the condition `dept='CS'`, and one based on the age condition (`age=XX` or `age<>XX`). The `uid`- and `age`-based noise values would change with each query, and so could be averaged away. The `dept='CS'` noise, however, would always be the same and could not be averaged away. The final averaged count would be perturbed by the `dept='CS'` noise layer.

Let's refer to this noise layer as the *static condition* noise layer (or just *static* noise layer for short).

The static noise layer prevents the difference attack *as executed above*: the queries on the left would have the `gender='M'` noise layer, while the queries on the right would not. Therefore the right and left answer would vary for every pair, not just the pair matching the woman's salary. The analyst, however, could simply observe that for every pair but one, the difference between the left and the right is the same. By way of example, suppose that the `gender='M'` layer always adds 2 to the count. For the buckets not matching the woman's salary, the `uid`- and `dept`-based noise layers are the same. The only difference is from the `gender='M'` layer and so for all these buckets the left answer is exactly 2 greater than the right answer. If this is not the case for one of the bucket pairs, then it can only be because the woman is in the right-hand bucket. Thus the analyst learns her salary. We call this the *first-derivative difference attack*, because the analyst is looking for a difference in the difference.

The first-derivative difference attack can be solved with one more per-condition noise layer. This noise layer is dependent on the combination of condition and distinct set of users. We call this the *dynamic condition* noise layer, or just *dynamic* noise layer for short. Now, because every bucket has a different set of users, the dynamic noise layer for `gender='M'` changes with each bucket. This forces a pseudo-random difference in the count of each bucket pair, thus preventing the attack.

5.3 Details of Layered Sticky Noise

Figure 4 illustrates the overall anonymization process in Diffix.

Each static noise layer is generated by seeding a PRNG with a hash of the following parameters concatenated together:

1. The column name (normalized, e.g. all lower case, unicode).
2. One of the following:
   (a) If the condition is an equality (= or <>), the value of the constant in the condition.
   (b) Otherwise if the condition is the result of a GROUP BY clause, the corresponding reported value,
   (c) Otherwise if the condition is a range, then the values of the two constants that define the range.
3. The condition operator (= or <> or 'range').
4. A secret salt that does not change from query to query.

Each dynamic noise layer additionally includes a value generated from the XOR of the hashes of the distinct `uid`'s.

The PRNG, so seeded, then generates a zero-mean value from a Gaussian distribution[5] with a standard deviation appropriate to the required amount of

[5] Distributions other than Gaussian may serve better, but in any event the noise is small and so we haven't yet explored this question.

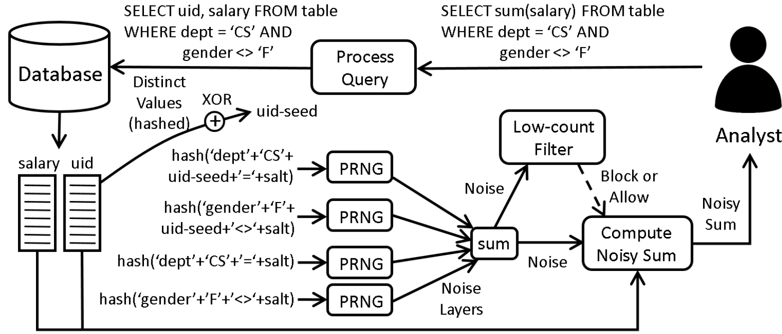

Fig. 4. Sticky Noise Layers and Sticky Noisy Threshold.

noise (see Sections 6 and 7). Note that all but the first of any repeated condition is removed from the query, and does not result in a new noise layer. In our current implementation, the standard deviation for each noise layer is $\sigma = 1$ for counting queries (counting distinct users). The standard deviation of the summed noise layers is then $\sigma = \sqrt{N}$, where N is the number of layers.

This is illustrated in Figure 4. The query requests the sum of salaries for men in the CS department. Diffix modifies the query to obtain the uid and salary columns from the database. The query has two conditions, `dept='CS'` and `gender<>'F'`. These are used to generate the seeds for each condition's corresponding static and dynamic noise layers. The four resulting noise layers are summed to produce a final noise value, which is in turn used by two functions: one that computes the noisy sum of salaries, and the low-count filter, described next.

5.4 Sticky Noisy Threshold

A valuable feature of any database system is the ability to discover the contents of columns. For instance, suppose a database contains the search strings that users submitted to a search engine. An analyst cannot know in advance the searches that have been made, and so would like to discover what the strings are. Diffix supports this feature through the GROUP BY clause, but only reports a given column value or combination of column values when at least some small number of distinct users have that value or combination of values. This feature

is called *low-count filter*: any response is silently suppressed if fewer than some small number of distinct users have that value.

This "small number" is not a hard-coded constant, but rather a *sticky noisy threshold*. Specifically, the threshold is sticky noise with mean K. Only if the number of distinct users that share a value exceeds the noisy value with mean K is the value reported.

In our implementation, $K = 4$, and the standard deviation of each noise layer is $\sigma = 1/2$. For example, if there are four noise layers (i.e. two each from two conditions), then the standard deviation is $\sigma = 1$. In addition, any bucket with two or fewer users is always filtered.

The reason we use a noisy threshold instead of a hard-coded threshold is that a hard-coded threshold K_h would allow an analyst to make a 100% confidence guess in those cases where the analyst knows that a given count of distinct users is either K_h or $K_h + 1$. The noisy threshold lowers an analyst confidence with high probability even when an analyst has this knowledge.

5.5 Layered Sticky Noise with Range Shifting Attacks

The layered noise mechanism described so far defends against difference attacks that remove conditions from a query. Ranges allow difference attacks that don't require removing a condition, but rather that simply adjust the range.

As an example, suppose that the analyst has access to a company database that includes entry and exit times for all employees. The analyst knows an employee (the victim) that left the building at 14:26 on a given day, that no other employees left 2 minutes before and after, and that the employee didn't again exit the building until the next day around 17:00. The analyst could then learn the salary of the victim with pairs of queries like the following:

```
SELECT count(*) FROM table
WHERE salary = 100000 AND
  exit >= '2015-06-17 14:27:00' AND
  exit < '2015-06-18 16:00:00'
```

```
SELECT count(*) FROM table
WHERE salary = 100000 AND
  exit >= '2015-06-17 14:25:00' AND
  exit < '2015-06-18 16:00:00'
```

Fig. 5. Range-based difference attack to learn the victim's salary. Subsequent pairs of queries would modify the range a small amount (`14:25:01, 14:25:02, ...`) so as to average away the range noise layers.

The query on the left excludes the victim while the query on the right includes the victim only if the salary is $100K. Note that the time ranges are large enough that the analyst is confident that there are enough users with that salary to ensure that the low-count filter is bypassed. With only a single pair of queries, the noise layers associated with the ranges cause the two queries to (probably) differ whether or not the victim is present. The analyst, however, can average away the range noise layers by incrementally modifying the range (e.g. '14:25:01',

'14:25:02', etc.) so that the noise is different but the set of users included in the range remains the same.

To defend against this, Diffix limits the granularity of ranges by "snapping" ranges into pre-defined sizes and offsets. Analysts are limited to these pre-defined *snapped alignments*. Diffix then adds static and dynamic noise layers based on the constants that define the boundaries of the range.

The basic idea of snapped alignment is that sizes are confined to an exponentially growing and shrinking set of values, and offsets fall on multiples of the size. For natural numbers, in our implementation we have chosen sizes from the set of values $\pm 1x10^N$, $\pm 2x10^N$, and $\pm 5x10^N$ where N is any positive or negative integer. This defines the sequence $[..., 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, ...]$ along with the corresponding negative numbers. Offsets are set at even multiples of the size, and are allowed to be shifted left or right by $1/2$ the size.

Resulting valid ranges include $[1, 2)$, $[20, 40)$, and $[0.01, 0.015)$. These can be shifted by $1/2$ to create valid ranges $[1.5, 2.5)$, $[30, 50)$, and $[0.0125, 0.0175)$. Invalid ranges include $[1.1, 2.1)$, $[30, 49)$, and $[0.01, 0.014)$.

There is nothing special about this specific choice of snapped alignment. They could just as well have been defined as powers of 2. We choose this "125" sequence because we believe that analysts will find it more natural than powers of 2. (Money typically comes in these denominations for the same reason.)

For datetimes, it is convenient to use the natural boundaries of for instance years, months, days, hours, minutes, and seconds. Within these units, we specify additional natural ranges. For instance, within years we further define 2, 3, 4, and 6 month intervals, and within days, we define 2, 4, 6, and 12 hour intervals.

In addition, Diffix imposes a maximum allowed range. The purpose is to prevent the analyst from averaging away the noise layers associated with a range by using a sequence of queries that grow the range. Diffix computes the maximum range by executing the anonymized min and max functions (Section 6.2), and then computing the smallest snapped range that encompasses this max and min. Any query whose range exceeds the maximum allowed range is rejected.

With snapped alignment, the above attack no longer works because the analyst cannot make a pair of ranges that on the one hand includes and excludes the victim, but on the other hand includes enough other users that the queries are not low-count filtered.

Of course there may well be cases where attacks with snapped ranges can in principle be formed. For instance, suppose that the victim happened to be the only employee to work on a given Sunday and didn't work on the following Monday. The analyst could then generate a query with a 2-day range of Sunday and Monday, and a second query only covering Monday. Both of these ranges adhere to snapped alignment, but nevertheless constitute a difference attack.

The noise layers associated with the range defends against this attack. The two queries would have a differently seeded noise layers, and so the answers may differ whether or not the victim is present.

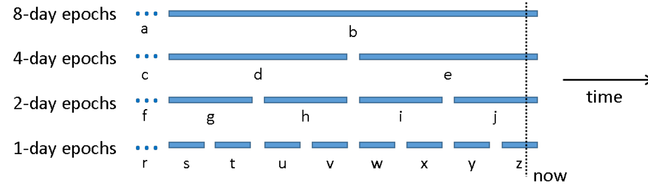

Fig. 6. Hierarchy of Time Epochs

5.6 Time Epochs for Difference Attacks on Changing Databases

All of the difference attacks described so far assume that the pair of queries operates on the same data. If the database changes (data is added, removed, or modified), and the analyst knows that the change is for only a single user, then the analyst can potentially execute a difference attack merely by executing the same query before and after the change.

To deal with this, Diffix defines a hierarchy of time epochs. Updates to the database after the last complete epoch are not included in answers. If for instance the smallest epoch is one day, then on any given day, only the database updates to the end of the previous day can be queried. For example, in Figure 6, a query made at the time labeled *now* would query the database as updated at the end of the day labeled y. For the hierarchy, each higher-level epoch may for instance double in length. So for instance the 1-day epochs (r - z) may fall under 2-day epochs (f - j), which fall under 4-day epochs (c - e), and so on.

One way to protect against this difference attack is to add two layers of noise for every epoch within which the query falls. So for instance the query done *now* would have layers of noise seeded by the epochs z, j, e, and b. As a result, relative to a query made at any previous point in time, there is a level in the hierarchy where on one hand a noise value differs, while on the other hand there are not enough differing noise values that the analyst can average away the noise. For instance, suppose that a single user change occurs during epoch s, and no change occurs between then and *now*. Even if the analyst queries every day and averages away the noise from the 1-day epochs, the 4-day epochs d and e would force a different noise value without being averaged away. This approach has the advantage of simplicity, but can result in a substantial amount of noise.

Another approach that doesn't require noise layers, but is more complex, is to dynamically determine which is the latest update that may be safely used. This can be done by finding a pair of adjacent epochs, at any level of the hierarchy, whereby the difference in distinct users between answers exceeds a noisy threshold. The query is then answered from the first epoch of the pair.

For example, suppose that the difference in distinct users between 1-day epochs s and t exceeds the noisy threshold. Later repeats of the query all give the same answer, that of epoch s. Now suppose that the difference between epochs x and y exceeds the noisy threshold. Subsequent to this, answers may be

given from epoch x. These answers will differ enough from the epoch s answers to hide changes due to any single user.

The use of higher-level epochs avoids the situation where the database changes slowly, but never enough to exceed the noisy threshold between adjacent epochs. Were this to happen, answers would become less accurate over time. Higher-level epochs captures these changes. For instance, suppose that a big enough change occurs between 1-day epochs s and t, but only small changes occurs afterwards. If the change between for instance 2-day epochs h and i exceeds the noisy threshold, then answers from epoch h may be released.

6 Anonymous Statistical Functions

To this point, we've described how we produce sticky noise, but not exactly how the noise is used in statistical functions. Currently our Diffix implementation can compute and add noise to the following statistical functions: count, sum, average, standard deviation, min, max, and median.

Of these, there are two types of statistical functions. One returns a value for an individual row (i.e. the value of a single user). These include min, max, and median, which (in the absence of noise) return the value of the single user with the min, max, or median value. We refer to these as *single-row* statistical functions. The other type of statistical function returns an answer that is a composite of all values. These include count, sum, avg, and stddev, and are referred to as *multi-row* statistical functions.

Stated informally, the goal of any anonymous statistical function in Diffix is to "hide" the presence or absence of any given user under the assumption that *the analyst knows all values except that of the given user*. For multi-row functions, Diffix uses two mechanisms: 1) masking outliers, and 2) adding enough noise so that there is substantial uncertainty as to the value for users with high values. For single-row functions, Diffix can return an exact value (i.e. the value of a row with no noise added). However, it does so only when there are enough distinct users with the same value. Otherwise, it returns a noisy value.

6.1 Anonymous Sum and Count

Because a count is simply the sum of rows, the same function is used for both sum and count. The function has two phases, a pre-processing phase and a summing phase. The input to the summing phase is:

1. A table with two columns, a *uid* column and a *value* column. The uid's are distinct, and the value contains each user's total contribution to the count or sum.
2. A "baseline" sticky noise value N_b derived from the noise layers. N_b has a standard deviation of $\sigma = \sqrt{N}$, where N is the number of noise layers (see Section 7). The actual noise is a multiplicative factor of N_b.

Depending on the function, the pre-processing phase produces the value as follows:

count(distinct uid): All values are '1'.
count(*): Each value is the number of rows for the uid.
count(column): Same as count(*), except rows with NULL values are not counted.
count(distinct column): Same as count(column), except duplicate values are removed before counting each user's rows.
sum(column): Each value is the sum of the values for the uid.
sum(distinct column): Same as sum(column), except duplicate values are removed before summing each user's values.

The summing phase consists of the following steps:

1. Generate two sticky thresholds T_1 and T_2.
2. Label the T_1 users with the highest values *group 1*.
3. Label the T_2 users with the next highest values *group 2*.
4. Define A_2 as the average of the values from *group 2*.
5. Replace the *group 1* values with A_2.
6. Sum the values, and add noise with magnitude $N_b * A_2$. Counts are rounded to the nearest integer.

If there are both positive and negative values, then the above computations are done separately for the positive and negative values, using lowest values instead of highest, and using a separate baseline noise N_b for the negative values. The actual added noise is the sum of the two noise values. Note that for count(distinct uid), $A_2 = 1$, and the noise has $\sigma = 2$.

This approach hides outliers both in terms of their direct effect on the true sum, and in terms of their effect on the added noise. By using groups of users from which to derive values, the effect of any one user is hidden in a crowd. By using noisy thresholds to select the groups of users, the uncertainty of the analyst is increased. Rounding adds uncertainty in rare cases where the analyst can learn and exploit the value of a single pair of noise layers.

Both the count(distinct column) and sum(distinct column) functions have a pre-processing step where duplicate values are removed. In this step, duplicates are removed in such a way as to maximize the resulting number of distinct uid's. This minimizes the contribution of any given user, allowing Diffix to minimize the noise.

Anonymous Average and Standard Deviation Both the anonymous average and the anonymous standard deviation use the anonymous sum function. The anonymous average is computed simply as the anonymous sum divided by the anonymous count.

To compute the anonymous standard deviation, first the true average is computed. Then, the square of the difference between each value and the true average is computed. Then the anonymous average of these squared differences is computed. Finally, the final output is the square root of this anonymous average.

6.2 Anonymous Min and Max

To compute the max value, Diffix computes the following two values, and selects the maximum of these two.

The first value is simply the largest value that has enough distinct uid's associated with it to pass the low-count filter.

For the second value:

1. Compute A_2 the same as with the sum function in Section 6.1.
2. Compute σ_2 as the standard deviation of *group 2*.
3. The second max value is computed as A_2 with noise $(N_b * \sigma_2)/8$.

This noise effectively eliminates the possibility that the analyst can reverse engineer the exact values that produced A_2 in the case where the values in *group 2* differ from each other. The divisor of 8 is chosen to minimize noise while still providing strong protection against reverse engineering values. Note that if all the values in *group 2* are the same, then $\sigma_2 = 0$ and no noise is added.

The anonymous min is computed the same as the anonymous max, but substituting low values for high.

The anonymous min and max functions can output the exact value for a given user. This value might even be the true min or max (though the analyst does not know this from the output). This does not reveal any information that the analyst could not otherwise have discovered with a query of the form:

```
SELECT column, count(*) FROM table GROUP BY column
```

and taking the highest (lowest) returned value as the max (min). On the other hand, the anonymous min and max functions may return a more accurate value than could be discovered using this query.

6.3 Anonymous Median

A similar approach is taken to compute the anonymous median. Specifically, the values from groups of distinct users above and below the true median are selected, and the anonymous median is computed from these values. The details are omitted due to space constraints.

7 Amount of Noise

Because Diffix does not "leak noise" with repeated answers as most differential privacy mechanisms do, individual query answers need have only enough noise to anonymize that given answer. Recall that the system is relatively more anonymous when the probability of a higher confidence guess is lower. While the decision of how much noise to add is up to the data owner, we believe that for sticky noise, a per-layer standard deviation of minimum $\sigma = 1$ provides adequate anonymity for counting queries.

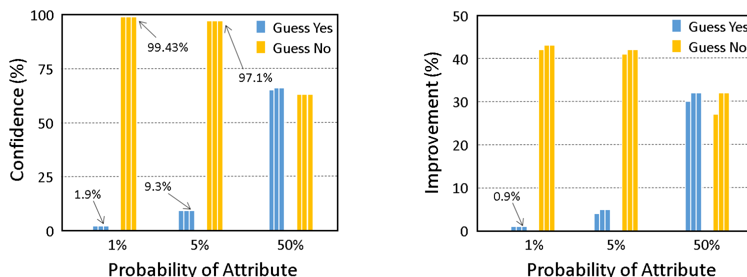

Fig. 7. Confidence C of an analyst's guess in a difference attack, and the improvement in confidence C_i over a statistical guess of the attribute probability P_a. Per-layer noise has $\sigma = 1$. Groups of four bars are for 4, 5, and 6 noise layers, left to right. Values are averages over 10^8 trials.

To support this claim, we provide the results of two simulations. The first simulates a difference attack on counting queries where the analyst has no a priori knowledge about the true count, and is able to exclude the victim with a single condition (as for instance with the example of Figure 3). Results (Figure 7) are given for between 4 and 6 noise layers for the query with the excluding condition, with a per-layer $\sigma = 1$. Answers are rounded to the nearest integer.

We define *confidence improvement* as $C_i = 100 * (C - P_a)/(100 - P_a)$, where P_a is the probability that a given attribute occurs in the population, and C is the analyst's confidence in a guess. For instance, if an attribute occurs in the population with $P_a = 60\%$ probability, and the analyst has $C = 80\%$ confidence in their guess, then the confidence improvement is $C_i = 50\%$.

In the simulation, if the answer for the query that conditionally includes the victim is greater than the answer for the query that excludes the victim, the analyst guesses that the victim has the attribute (i.e. a given salary). The results are shown in Figure 7 for the cases where the attribute probabilities are $P_a = 1\%$, $P_a = 5\%$, and $P_a = 50\%$.

Starting with the $P_a = 50\%$ case, the difference attack improves the analyst's confidence to between $C = 62\%$ and $C = 66\%$. While somewhat better, the analyst's confidence remains acceptably low (in our opinion). Now consider the case where $P_a = 1\%$. In the case where the analyst guesses that the victim has the attribute, the analyst's confidence is $C = 1.9\%$, a 0.9% improvement. A guess that the victim does not have the attribute improves confidence by $C_i = 43\%$, from $P_a = 99\%$ statistical confidence to $C = 99.43\%$. Put into the perspective of the difference attack of Figure 3, assuming that the probability of the victim having any given salary is 1%, the analyst could improve his or her confidence

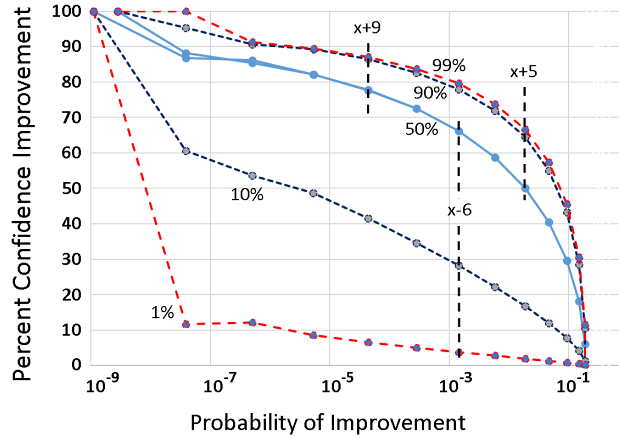

Fig. 8. Improvement in analyst's confidence C_i, and probability of getting that improvement, when analyst knows the true answer must be one of two possible known values (x and x+1). Noise is 0 mean Gaussian with standard deviation 2. Three curves are for cases where lower value x occurs with 1%, 10%, and 50% probability. Points to the left represent more noise. Points where noisy count is x-6, x+5, and x+9 are labeled.

in what the victim's salary is not, but still has very little confidence in what the victim's salary is.

The second simulation examines the case where the analyst knows through external knowledge that a given count is one of two possible values, x or $x+1$. In this case, the analyst's confidence in the correct answer is higher when the noisy answer is further from the two possible counts. For instance, suppose that the analyst knows that the only two possible answers are 20 and 21, and that they occur with equal probability. If the noisy answer is 21, then the analyst's confidence that the true answer is 21 is only $C = 53\%$ ($C_i = 6\%$). If the noisy answer is 26, however, the analyst's confidence that the true answer is 21 is $C = 79\%$ ($C_i = 59\%$). On the other hand, the probability of getting a noisy count of 26 is much smaller than getting 21 (0.06% versus 19%).

Figure 8 plots the confidence improvements and corresponding probability of improvement for the cases where the probability of the lower value (x) occurring is $P_a = 1\%$, $P_a = 10\%$, and $P_a = 50\%$. In this simulation, the analyst guesses x if the noisy count is x or smaller, and guesses x+1 otherwise. Each simulation tested 10^{11} trials. The left-most data points represent an average of a few 10's of trials, the right-most data points correspondingly more.

The data points on the lower right represent the case where the noisy count is either x or x+1. This happens with a probability of roughly 36%. In these cases, the improvement in the analyst's confidence beyond the statistical probability of the expected outcome is very little.

Consider the point on the $P_a = 50\%$ curve where the noisy count is x+9. Here the analyst's confidence improvement is $C_i = 78\%$, but this improvement happens in only 1 out of roughly 23K trials. From the figure, we can see that noise with as low as $\sigma = 2$ provides good anonymity. Obviously more noise would provide still better protection, but we don't believe more noise is necessary. Note that, as a rule, an effective attack requires several conditions, and therefore we are not too concerned with queries that have noise less than $\sigma = 2$ (i.e. only one condition).

8 Summary, Limitations, and Future Work

This paper presents a description of Diffix, a database anonymization system that is easy to configure, is strongly anonymous, allows unlimited queries, adds minimal distortion to answers, and supports a variety of useful statistical functions.

In the process of describing Diffix, this paper shows how Diffix defends against certain basic attacks. Due to space limitations, this paper does not provide a thorough analysis of the attacks and defenses currently known to the authors. This is a limitation of the paper.

A major limitation of the work overall is the fact that we do not have a formal treatment of Diffix. As a result, while we are not aware of any attacks that break Diffix's anonymity, we cannot make a positive statement that no such attacks exist. Moving forward, we hope to formalize and prove Diffix as much as possible. It is not clear how far we can get with a formal approach, so in parallel we plan to deploy an implementation of Diffix and make it available for attack, possibly with prizes for successful attacks. This exercise can lead to improvements to Diffix, a better understanding of how to evaluate risks (should some attacks be possible), and an overall increase in confidence in this new technology.

A big open question is whether Diffix provides enough utility to be used widely in practice, and if not, how it's utility may be improved. Is the accuracy adequate? Do analysts need richer query semantics? Do analysts need more statistical functions, and if so can these be provided? How much does the inability for analysts to "see the data" impede the analytic process overall?

We expect to research these and many other questions in the future, and hope that Diffix inspires the broader research community to pursue this and other new architectures.

References

1. Article 29 Data Protection Working Party Opinion 05/2014 on Anonymisation Techniques . http://ec.europa.eu/justice/data-protection/article-29/documentation/opinion-recommendation/files/2014/wp216_en.pdf .
2. D. E. Denning. Secure statistical databases with random sample queries. *ACM Transactions on Database Systems (TODS)*, 5(3):291–315, 1980.

3. D. E. Denning, P. J. Denning, and M. D. Schwartz. The tracker: A threat to statistical database security. *ACM Transactions on Database Systems (TODS)*, 4(1):76–96, 1979.
4. I. Dinur and K. Nissim. Revealing information while preserving privacy. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 202–210. ACM, 2003.
5. C. Dwork. Differential Privacy. In *ICALP*, 2006.
6. Ú. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1054–1067. ACM, 2014.
7. I. Fellegi and J. Phillips. Statistical confidentiality: Some theory and application to data dissemination. In *Annals of Economic and Social Measurement, Volume 3, number 2*, pages 399–409. NBER, 1974.
8. I. P. Fellegi. On the question of statistical confidentiality. *Journal of the American Statistical Association*, 67(337):7–18, 1972.
9. F. Prasser and F. Kohlmayer. Putting statistical disclosure control into practice: The ARX data anonymization tool. In *Medical Data Privacy Handbook*, pages 111–148. 2015.
10. Waltraut Kotschy. The new General Data Protection Regulation - Is there sufficient pay-off for taking the trouble to anonymize or pseudonymize data? https://fpf.org/wp-content/uploads/2016/11/Kotschy-paper-on-pseudonymisation.pdf, Nov. 2016.