Journal of Computer Science

Original Research Paper

Randomization of Statistical Queries of Type Median: A Simulation Approach

[1]Jose Daniel Velazco, [2]Mohammed Awad and [3]Ernst L. Leiss

[1]*Department of Computing Sciences, University of Houston-Clear Lake, Houston, TX 77058, USA*
[2]*Department of Computer Science and Engineering, American University of Ras Al Khaimah, UAE*
[3]*Department of Computer Science, University of Houston, TX 77204, USA*

Article history
Received: 27-09-2017
Revised: 28-12-2017
Accepted: 13-01-2018

Corresponding Author:
Mohammed Awad
Department of Computer Science and Engineering, American University of Ras Al Khaimah, UAE
Email: mohammed.awad@aurak.ac.ae

Abstract: Researcher and third party access to data pertaining to individuals is becoming the norm. The conclusions drawn from such data can be extremely beneficial. However, data owners must maintain the secrecy of the sensitive data fields and make sure it is protected against inference attacks. There are several techniques and restrictions that can be made on queries to prevent adversaries from inferring and identifying sensitive data related to specific individuals. One of the proposed techniques to prevent the disclosure of private data is randomization. In this study, we demonstrate and analyze the implementation of randomization in statistical queries of the selector function median and the results of an extensive simulation. The randomization technique yields a possibly erroneous yet usually reasonably accurate response to every query. In addition, the inference procedure is explained and potential modifications to counter the randomization technique are analyzed and tested against it. We show that, despite these modifications, randomization protects the data by adding uncertainties into the inference procedure, thus, maintaining differential privacy. The results of an extensive simulation testing the various parameters of the randomization technique on randomly generated databases are shown and explained.

Keywords: Inference Attacks, Statistical Database Security, Median Queries, Randomization

Introduction

Even though data owners are likely to hide private information (such as a patient's disease, an employee's salary, or a student's grade) before granting database access, an adversary may in one way or another be able to infer some of that private hidden data and deduce sensitive private information about specific individuals. Back in 2006, AOL released 20 million search keywords for 650,000 of its users for research purposes. Despite masking user identity, several were identified (Heatherly *et al*., 2013). Three days later, AOL took down the published database, but it was already mirrored on other websites. Such an invasion of privacy resulted in a lawsuit against AOL and several concerns about user privacy and the impact of inference attacks. Nowadays, the amount of data gathered by the government and the private sector has significantly increased and the impact of inference attacks is more significant than ever (Naveed *et al*., 2015).

The idea of protecting sensitive (private) information is, by itself, important. However, it extends further than that. By law, it is mandatory to protect some individuals' private information, for example medical records (Leiss, 1982a). Compromising an individual's record not only is a problem for the individual, but a legal issue for the database owner. These risks complicate the use of statistical information as it is possible to infer data from a combination of legitimate statistical information to obtain access to private data, known as inference attacks (Leiss, 1982a). Adversaries may be able to use a set of linear models to relieve some of the database confidential attributes (Sarathy and Muralidhar, 2002; Cynthia, 2006). Currently, there are methods used to prevent inference attacks by rejecting statistical queries that, in combination with previous queries, can compromise the value of an element in the database (Adam and Worthmann, 1989). However, these methods come with significant disadvantages. Primarily, the database must ensure that the current query's response does not yield a compromise when combined with previous queries, which adds to the work done by the database. On top of that, it may be possible for multiple users to work together and yield a compromise. Some more sophisticated techniques have been found to audit statistical queries and determine whether new queries would yield compromises (Cavallo and Canfora, 2016; Lu *et al*., 2015). Additionally, assuming the database checks queries among users, the database would eventually have to reject every single query requested (Leiss, 1982a). Other masking techniques involve random noise addition either to the query itself or the query output (Giggins and Brankovic, 2012; Hegadi *et al*., 2011), while other techniques ensure perturbation via microaggregation (Muralidhar and Sarathy, 1999). Some perturbation techniques consist in swapping query elements based on distributions calculated from the database elements (Zou and Zhang, 2012).

Note that while data owners do hide private data for individual records, they may still allow statistical queries, which return a value based on these private elements. For example, such a query may return the average or the median value. Our paper's main focus is statistical queries of type median. We propose a solution that does not require the database to check for potential compromises. This approach is relatively simple to implement for an existing database and probabilistically protects a database from successful inference attacks. We call this solution randomization. Essentially, randomization adds a random element from the database to every query and provides a possibly erroneous yet usually reasonably accurate response (Leiss, 1982a). This solution differs significantly from random noise addition in that we use actual values from the database. When adding random noise there is always the danger that it can be removed if the random generator is known, which is usually unavoidable since only the seed is unknown, but the algorithm is known. However, in our approach, the randomly selected database element is chosen from a range of numbers relative to the true median of the query, which mitigates that danger.

In this study, we show the implementation of randomization in statistical queries of the selector function median and the results of an extensive simulation. It is worth mentioning that in the case of medians, noise addition is likely to change nothing unless that noise is added to the actual median value that is returned; adding noise to the other values will most likely not change the value returned because of the properties of the median function (unlike the mean). Thus, such an approach (randomization for queries of type median) should maximize the accuracy while minimizing the inference risks and eventually improve differential privacy (Muralidhar and Sarathy, 2003).

Background

A statistical query is a statistical question made to some set of data, e.g., average, median, max, min, etc. (Leiss, 1982a). In particular, we will be looking at statistical queries of type median. The query notation is based on the key-specified model (*DK*). The *DK* model, as described in (Leiss, 1982a), goes as follows:

$$DK : \{1, \ldots, N\} \rightarrow \mathbb{R}$$

where, N is the number of elements in the database. Each element in the database is associated with a private value. This is the value we are trying to protect. This association of element and value is denoted in the following form: s is an index to the database and $DK(s)$ is the private value associated with s. Each query requires a set of exactly k elements for a fixed $k \geq 2$. The *DK* model requires that every requested query be of same length. In addition, every statistical query yields a result. The result depends on the type of function requested for the query. In our case, we will use the selector function median. By selector function, we mean that the response to the query must be one of the database elements. The function is denoted as $f(Q)$ where Q is the set of database elements involved. For example, the function $f(DK(i_1), DK(i_2), \ldots, DK(i_n))$ for a function of type median yields the value $DK(s_i)$ where s_i is the index associated with the median value of the elements included in the query. We will use this notation throughout to explain our procedures.

In the past, we have shown the implementation of randomization of statistical queries of type average (Leiss, 1982a; 1981; 1982b). Statistical queries of type average can infer data by using the query elements and the query response in combination with the previous queries and responses to build a system of linear equations and solve for each of the elements involved in the queries (Leiss, 1981). The results obtained from this simulation showed that the use of randomization, i.e., randomly adding an element as part of every query and altering the responses slightly, protects the data from inference attacks (Leiss, 1981). The error associated with every response builds up when solving the system of linear equations, which means that the inference attacker will obtain incorrect results. More specifically, the error of the compromised elements is equal to the error of the responses (rather small), multiplied by the condition number of the matrix involved in the compromise (solving the system of linear equations). Since this condition number can be shown to be larger than k, the error of the compromised elements is large. We want to apply the same principle to queries of type median. However, there are major differences between the two. First of all, the queries of type median are selector functions whereas averages are not and may yield responses not present in the database. More importantly, every selected value for queries of type average affects the result of the response. Queries of type median may obtain many different randomly selected values that yield the same response. For example, suppose we have a set of values {0, 0, 0}. The average and median responses are both equal to 0. Now suppose we randomly select a value (from the database) and the said value is 1. The average value is now 0.25, but the median value is still 0. Now suppose we randomly select another value (from the database) and the said value is 40. The average value is now 10, but the median value is still 0! The implementation of randomization for queries of type average does not work directly for queries of type median for the reasons described above. Therefore, we had to find a different implementation involving the same principle and ensure that we maintain the same security as queries of type average.

In our research, we present the results of a simulation of statistical queries of type median after applying the proposed randomization technique. The queries were made with the intention to compromise database elements using inference attacks.

Methodology

In this section, we explain statistical queries of type median and their inference procedure. Then, we apply the randomization methodology on such queries. After that, we propose and implement modifications to eliminate the previously witnessed inference attacks.

Statistical Queries of Type Median

A statistical query of type median yields the value situated in the middle of a set of sorted data. Note that there are two possible queries. The query may be of odd size. In this case, the result is exactly the value in the middle. On the other hand, the query may be of even size and we must choose one of two conventions. We either choose the smaller or larger median. Alternatively, we would find the average of the two medians. However, we are dealing with selector functions and such a value does not necessarily exist in the database. Therefore, we consistently choose one of the two conventions. Either way, the implementation of the inference procedures and randomization implementation we propose is not affected by the chosen convention (Leiss, 1982a). For simplicity, we will establish the value of k, the size of the query, to be odd.

Inference Procedure for Medians

Before we explain our security approach, we will show the procedure we used to compromise a database. This procedure is proved and formally described in (Leiss, 1982a). We will be looking at a small example to demonstrate the procedure on a sample database. We will make a sample database comprised of distinct values as it is the worst-case scenario. In other words, no two indices in the database may have the same private value. Take the following sample database (Table 1).

Table 1: Sample database (Example I)

s	DK(s)
1	3
2	5
3	1
4	7
5	4

We select the size of our queries to be of size $k = 3$. Note that k must be at least 2 less than the size of our database. In other words, $k + 2 <= N$ is needed to achieve a compromise. The procedure requires us to initially request $k + 1$ queries using k of the same $k + 1$ indices. For simplicity, we will use the first $k + 1$ indices. We get the following queries and responses:

$$f(1, 2, 3) = f(DK(1), DK(2), DK(3)) = f(3, 5, 1) = 3$$
$$f(1, 2, 4) = f(DK(1), DK(2), DK(4)) = f(3, 5, 7) = 5$$
$$f(1, 3, 4) = f(DK(1), DK(3), DK(4)) = f(3, 1, 7) = 3$$
$$f(2, 3, 4) = f(DK(2), DK(3), DK(4)) = f(5, 1, 7) = 5$$

The first column is supplied by the query requester and the fourth column is the response to the corresponding query. The intermediate columns are there to explain the results. From this point on, we will omit the first column since the second column includes the exact same information. Notice we get exactly two different medians, one smaller than the other. For simplicity, we will call the smaller median g and the larger median h:

$$g = 3, h = 5$$

We can now split our indices into two different sets, *a* set of indices G, whose values are less than or equal to g and a set of indices H, whose values are greater than or equal to h. One can see that the lower median is obtained when a higher value is missing and a higher median is obtained when a lower value is missing. Therefore, G will contain the indices missing from the queries producing the median h and H will contain the indices missing from the queries producing the median g. We get the following sets:

$$G = \{1,3\}, H = \{2,4\}$$

Although we don't know the value of any index, we can tell that either the index 1 or 3 is associated with the private value of 3. We can make a similar observation about the set of H. Next, we create a set G' that contains the indices of G minus any two indices of G. In the example, we get:

$$G' = \{\}$$

Now, we form the set G' combined with H and our s_{k+2} index and get:

$$G' \cup H \cup \{s_{k+2}\} = \{2,4,5\}$$

The set contains all previously used indices except for two low indices and a new added index. The median obtained from this set will help us determine if the added index, s_{k+2}, is an index associated with a higher or lower value than h. Note that it is not possible for it to be equal to *h* because our database contains distinct values and that value is associated with some index in *H*. We now find the median of this query and get:

$$f(DK(2), DK(4), DK(5)) = f(5,7,4) = 5$$

Because we obtained *h* as our result, it implies that the new added element is associated with a lower value than *h*. Therefore, we can make the following observation:

$$DK(5) < h$$

The alternative would be that $DK(5) > h$ if and only if the query's result would have been a value higher than *h*. The following step depends on this condition. Since $DK(5)$ is less than *h*, we need a subset of *H* of the same length minus one, H_0. We choose any element of *H*, remove it and get:

$$H_0 = \{2\}$$

We could have chosen $H_0 = \{4\}$. The result would yield a compromise as well. We also need a set formed by the union of *G* and the s_{k+2} index. We get:

$$G \cup \{S_{k+2}\} = \{1,3,5\}$$

With these two sets, we will form the remaining needed queries. Every query is made up of the elements in H_0 and *p* elements of $G \cup \{s_{k+2}\}$, where $p = (k+1)/2$. We use every combination of *p* elements in $G\cup\{s_{k+2}\}$ with H_0 to make $p+1$ queries and get:

$$f(DK(1), DK(2), DK(3)) = f(3, 5, 1) = 3$$
$$f(DK(1), DK(2), DK(5)) = f(3, 5, 4) = 4$$
$$f(DK(2), DK(3), DK(5)) = f(5, 1, 4) = 4$$

We obtain two medians as well. One median occurs *p* times and the other occurs only once. The median that occurs p times is the private value associated with the missing index in the query that obtained a unique median. As a result, we infer $DK(5) = 4$, which is indeed true.

If we had obtained $DK(5) > h$, we would do the same with the sets G_0 and $H\cup\{s_{k+2}\}$ instead. The procedure takes at most $3(k+1)/2+2$ queries to compromise an element (Leiss, 1982a). However, the compromised element cannot be chosen beforehand. It is possible that repeated procedures yield the same compromise. In addition, there are elements that are safe from compromise since they are never the response to any query. The safe elements are those that, once sorted, are located in the $k/2+1$ ends of the sorted set (Leiss, 1982a). Either way, once an element has been compromised, the database is said to be compromised (Leiss, 1982a).

Randomization Approach for Medians

With averages, we added an element to the query and returned as a response the average of such query. With medians, we will return, when possible, a different value in the database based on the range of the previous and next element relative to the median in the query. Figure 1 shows the possible responses to a query when randomization is applied. The possible responses are denoted as follows:

- *m*: True median of original query
- *n*: next sorted element relative to median in query ($n>m$)
- *p*: Previous sorted element relative to median in query ($p<m$)
- *i*: An element in the database but not in the query whose value is between *p* and *m* ($p < i < m$)
- *j*: An element in the database but not in the query whose value is between *m* and *n* ($m < j < n$)

There are also two "gaps". The gap between *p* and *m* and the gap between *m* and *n*. The values of *I* and *J* are the magnitudes of the gaps:

$$I = m - p,\ J = n - m$$

We will see that the most secure queries are those capable of returning an *i* or *j* value. However, sometimes a value for either *i* or *j* cannot be found. Because we choose random elements from the database, it is possible we do not find a value within the gaps since such a value may not even exist in the database. Therefore, we establish a tolerance number. The number is used to determine when should we stop looking for an *i* or *j* element. For instance, let the tolerance number be 20. If we request 20 random elements, one at a time and none of them are an *i* or *j* value, we continue as if there was no such value. This brings up a question about security vs. performance. Evidently, the higher the tolerance the more likely one is to find said value. However, the higher the tolerance, the more work the database must perform. The tolerance number can be determined by the database manager. We show in the results section the differences in security between giving an *i* or *j* value vs. an *m*, *n*, or *p* value. Based on said results, the manager can decide the best option for the database.

The randomization process is quite simple and there are only four different cases.

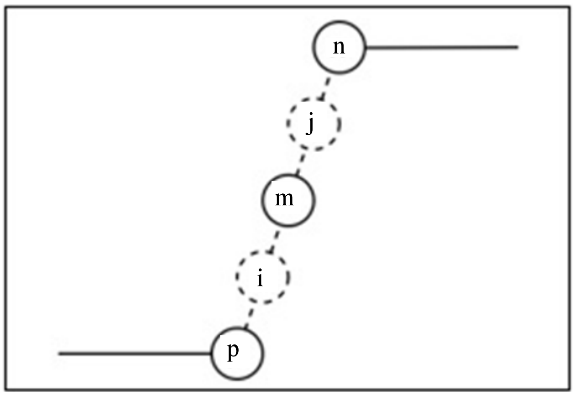

Fig. 1: Possible responses

Case 1: I = J = 1

The gaps are both equal to 1 implying that there is no possible value for *i* or *j* in the database (recall that we assume that no entries occur repeatedly). Therefore, we return *m*.

Case 2: I > J

In this case, the *I* gap is larger than *J* and, by default, larger than 1. A randomly selected element in the database whose value lies in the *i* range is given as the query's response. If no value is found, the value of *p* is given instead.

Case 3: I < J

Similarly, a randomly selected element in the database whose value lies in the *j* range is given as the query's response. If no value is found, the value of *n* is given instead.

Case 4: I = J > 1

Both gaps are of same magnitude. A randomly selected element in the database found whose value is either in the *i* or *j* range is given as the query's response. If no element is found, the value of *m* is given instead.

It is important to note why case *I* = *J* = 0 cannot occur. For this randomization study, we are assuming that the selected values are pairwise distinct. It definitely makes sense methodologically since we want to test how far this approach can take us towards guaranteeing security (inference control). Obviously, if all elements are identical, randomizing of any kind will not achieve anything (except that there might be some uncertainty about the *k*/2–1 smallest and the *k*/2–1 largest elements in the database). While the "pair-wise distinct" assumption may not hold universally, it is not unreasonable to assume that for many queries it does hold (note we use only *k* + 2 elements drawn from the database and these are likely to be pairwise distinct) – and therefore it establishes a baseline.

The error involved in the queries is at most the next or previous sorted element in the query. As a result, the error is relatively and reasonably small. The randomization technique may look odd at first. For instance, why choose the biggest gap and not the smallest gap? More likely, the smallest gap would provide a more accurate response. The inference procedure is quite robust. Our goal is to confuse the inference procedure by having some indices associated with values less than or equal to g be put into the *H* set and vice versa. In other words, we want to introduce errors into the procedure so that the user is uncertain of the obtained results. To demonstrate this, take the following example of what we claim to be the worst-case scenario. In this scenario, we make our values to be consecutive. Consider the following sample database (Table 2):

We let our query size be *k* = 3 and compute *p* = (3+1)/2 = 2. We let our *k*+1 indices be the first *k*+1 indices and obtain the following queries, gap magnitudes and responses:

$$f(DK(1), DK(2), DK(3)) = f(1,2,3) = 2\, I = 1,\ J = 1\ I = J = 1 \text{ Case } 1$$
$$f(DK(1) DK(2), DK(4)) = f(1,2,4) = 3 \text{ or } 4\, I = 1,\ J = 2\ I < J \text{ Case } 3$$
$$f(DK(1), DK(3), DK(4)) = f(1,3,4) = 2 \text{ or } 1\, I = 2,\ J = 1\ I > J \text{ Case } 2$$
$$f\big(DK(2), DK(3), DK(4)\big) = f(2,3,4) = 3\, I = 1,\ J = 1\ I = J = 1 \text{ Case } 1$$

We applied randomization as described in the four cases. Notice the first and fourth queries have gaps of 1 and we have no room to apply randomization. Thus, the true median value was given as the response. The second and third query may yield one of two possible responses. For instance, the second query may return the value of 3 if and only if the value of 3 was randomly selected within our tolerance number. Otherwise, we return the value of 4. Similarly, we get the same scenario for the third query. As a result, we obtain four possible different combinations from these responses. Note that the inference attacker would only get one of these four combinations. Let's suppose we get the scenario in which we obtained the medians 2, 3, 2, 3 respectively. In this case, we follow the procedure to determine *g*, *h*, *G* and *H*:

$$g = 2, h = 3$$
$$G = \{1,3\},\ H = \{2,4\}$$

We make our sets *G*' and $G' \cup H \cup \{s_{k+2}\}$:

$$G' = \{\}$$
$$G' \cup H \cup \{s_{k+2}\} = \{2,4,5\}$$

We make the query using the $G' \cup H \cup \{s_{k+2}\}$ elements and get:

$$f(DK(2), DK(4), DK(5)) = f(2,4,5) = 2\, or\, 3$$
$$I = 2,\ J = 1\ I > J\ Case\ 2$$

Table 2: Sample database (Example II)

s	DK(s)
1	1
2	2
3	3
4	4
5	5

At this point, we can obtain one of two different responses. If we get the value of 3, we make the observation $DK(5) < 3$ since we got the value of h as a result. If we obtain the value of 2, the inference attacker now knows something is wrong, as it is not possible to obtain a lower value than h from this query. The inference attacker has two options when this happens. The attacker may now be uncertain of the procedure and stop, or the attacker may decide to proceed as normal and hope for a compromise. We'll assume the worst case and proceed. Because 2 is less than h, we conclude $DK(5) < 3$. In this case, both results yield the same outcome. Thus, we proceed the same way regardless of the response received. We now make our H_0 set and get:

$$H_0 = \{2\} or\, H_0 = \{4\}$$

Previously, we mentioned that the set H_0 is the same as the set H minus any one of its elements. Since we choose one value and remove at random, it is possible we obtain one of the two sets above. One important thing to note is that the subset $H_0 = \{2\}$ is erroneous (the inference attacker does not know that of course) because the set of H presumably only contains indices associated with high values. However, we know that $DK(2) = 2$ which is not greater than or equal to h. This is an error introduced into the inference procedure. The alternative is $H_0 = \{4\}$ which is indeed a correct subset of H since we know $DK(4) = 4$ and is greater than or equal to h. We will proceed with both cases and show that despite of being possible to obtain a correct subset, the procedure can still yield incorrect results. Let's suppose we get $H_0 = \{2\}$. We form our $G\cup\{s_{k+2}\}$ set:

$$G \cup \{s_{k+2}\} = \{1,3,5\}$$

Following the procedure, we request our p+1 queries and get:

$f(DK(1), DK(2), DK(3)) = f(1, 2, 3) = 2, I = 1, J = 1, I = J$, Case 1
$f(DK(1), DK(2), DK(5)) = f(1, 2, 5) = 3$ or 4 or 5, $I = 1, J = 3$, $I < J$, Case 3
$f(DK(2), DK(3), DK(5)) = f\,2, 3, 5) = 4$ or 5, $I = 1, J = 2$, $I < J$, Case 3

There are multiple combinations of responses we may obtain from these queries. We know we must get one median that occurs p times and a second median that occurs only once. To analyze our results, we will classify our results as a successful procedure or a successful compromise. A successful procedure is an inference procedure, which produces a result. A successful compromise is a successful procedure whose result is a correct compromise. Note not all successful procedures are successful compromises, but all fail procedures are also fail compromises. For instance, if we obtain three different medians or we do not get two medians with the required occurrences, we claim the inference procedure to be a failed procedure and fail compromise. Table 3 contains a list of combinations of the queries' responses in the same order as the queries and whether we obtain a successful inference procedure and compromise.

From the six possible responses, four of them are said to be a fail since no value is obtained from the procedure. The other two are said to be successes since they produce a result. However, only one of the two successes is a successful compromise. In other words, there is a 16.67% chance that the inference attacker receives a correct compromise for this database and this value of k. Even then, the observation we made earlier stated that $DK(5) < 3$ and our result says that $DK(5) = 5$ (which is correct). The two contradict each other. Can the inference attacker accept the result even with this uncertainty? What if the user instead received $DK(5) = 4$ on top of the uncertainty? We show that as we get more complicated and larger databases and use smaller sizes of k relative to the database size (which are more realistic), the probability of successful compromises decreases and the level of uncertainty increases.

The alternative was to get $H_0 = \{4\}$. We obtain the same set $G\cup\{s_{k+2}\} = \{1, 3, 5\}$ and request the following queries:

$f(DK(1), DK(3), DK(4)) = f(1, 3, 4) = 1$ or 2, $I = 2, J = 1$, $I > J$, Case 2
$f(DK(1), DK(4), DK(5)) = f(1, 4, 5) = 1$ or 2 or 3, $I = 3$, $J = 1$, $I > J$, Case 2
$f(DK(3), DK(4), DK(5)) = f(3, 4, 5) = 4, I = 1, J = 1, I = J$, Case 1

We repeat the same analysis with the possible combinations we obtain from the p+1 queries and get the following results (Table 4).

We produce the exact same results for a different index. We may have obtained a valid subset of H for H_0. However, the indices in G are used as part of the queries and we know G contains incorrect indices as well.

These are the results for the first combination of responses from the first k+1 queries. Let's look at the second combination of 2, 4, 2, 3. With these responses, we do not obtain two unique medians. However, one can still divide the indices evenly into two sets. We can change the value of g and h to be inequalities instead and claim that:

$$g \leq 2,\ h \geq 3$$

Table 3: Queries responses and compromise status

Responses	Procedure	Result	Compromise
2, 3, 4	Fail	None	Fail
2, 3, 5	Fail	None	Fail
2, 4, 4	Success	$DK(5) = 4$	Fail
2, 4, 5	Fail	None	Fail
2, 5, 4	Fail	None	Fail
2, 5, 5	Success	$DK(5) = 5$	Success

Table 4: P+1 Queries responses and compromise status

Responses	Success	Result	Compromise
1, 1, 4	Success	$DK(1) = 1$	Success
1, 2, 4	Fail	None	Fail
1, 3, 4	Fail	None	Fail
2, 1, 1	Fail	None	Fail
2, 2, 4	Success	$DK(1) = 2$	Fail
2, 3, 4	Fail	None	Fail

Following the same procedure, the indices missing in the queries that obtained a response of 3 or higher go into the set of G and the indices missing in the queries that obtained a response of 2 or lower go into the set of H. We get the following sets:

$$G = \{1,3\},\ h = \{2,4\}$$

Notice both sets contain the same elements as our first scenario! We will not proceed with the steps since the procedure would yield the same responses. If we follow the procedure on the two remaining scenarios, we will obtain the same sets and responses. In addition, we have an extra uncertainty since we obtained 3 different medians instead of 2. In the next section, we explain how an inference attacker could go about when obtaining results that do not go accordingly to the standard inference procedure.

Earlier in the section, we mentioned how some elements are protected from compromise, as they are never the response to any query. Because of randomization, those elements that used to be safe from compromise are not necessarily completely safe anymore. This can be observed by looking at the uncertain compromise of $DK(1) = 1$. Before, with a size of $k = 3$, the value of 1 would never be a possible response of any combination of 3 elements from the 5 element database. However, when we choose to select the previous element relative to the median, we now use that element as part of a response. Either way, our goal is not to protect specific elements, but to ensure that the database is overall secure.

Modifications to the Inference Procedure

There were simpler randomization implementations that successfully broke the standard inference procedure. However, simple changes to the procedure would yield a successful compromise. One modification was shown earlier. We changed the values of g and h to be inequalities instead. However, it gets more complicated than that. Our database and query sizes were small. For larger values of N and k, we obtain results that are not as friendly. For instance, for $k = 7$, we could get 6 repeated medians and 2 different medians for our initial k+1 queries. To explain these modifications, suppose we do not apply the proposed randomization technique. Instead, we apply a similar technique as the one described in (Garcia *et al*., 2010): Instead of adding a random element from the database, we will remove the median value of the query and return the median of the new query. Consider the following sample database (Table 5).

We let our size of $k = 5$ and compute $p = (5+1)/2 = 3$. We produce our k+1 queries and get the following responses:

f(*DK*(1), *DK*(2), *DK*(3), *DK*(4), *DK*(5)) = *f*(4, 2, 1, 8, 9) = *f*(1, 2, ~~4~~, 8, 9) = 8
f(*DK*(1), *DK*(2), *DK*(3), *DK*(4), *DK*(6)) = *f*(4, 2, 1, 8, 6) = *f*(1, 2, ~~4~~, 6, 8) = 6
f(*DK*(1), *DK*(2), *DK*(3), *DK*(5), *DK*(6)) = *f*(4, 2, 1, 9, 6) = *f*(1, 2, ~~4~~, 6, 9) = 6
f(*DK*(1), *DK*(2), *DK*(4), *DK*(5), *DK*(6)) = *f*(4, 2, 8, 9, 6) = *f*(2, 4, ~~6~~, 8, 9) = 8
f(*DK*(1), *DK*(3), *DK*(4), *DK*(5), *DK*(6)) = *f*(4, 1, 8, 9, 6) = *f*(1, 4, ~~6~~, 8, 9) = 8
f(*DK*(2), *DK*(3), *DK*(4), *DK*(5), *DK*(6)) = *f*(2, 1, 8, 9, 6) = *f*(1, 2, ~~6~~, 8, 9) = 8

Because our queries are of even size, we choose the higher value median as our query's response. At this point, it is impossible to divide the indices evenly in two different sets. However, we can still choose our value of g and h to be 6 and 8 respectively. We get the following sets:

$$\begin{aligned}
&G = \{1,2,3,6\}\\
&H = \{4,5\}\\
&G' = \{1,2\}\\
&G \cup H \cup \{s_{k+2}\} = \{1,2,4,5,7\}
\end{aligned}$$

As one can see, the set of G is much larger than the set of H. This will bring complications towards the end of the procedure. For now, we proceed as normal and request the following query:

f(*DK*(1), *DK*(2), *DK*(4), *DK*(5), *DK*(7)) = *f*(4, 2, 8, 9, 5) = *f*(2, 4, ~~5~~, 8, 9) = 8

We make the observation $DK(7) < 8$. We form the following sets:

$$\begin{aligned}
&H_0 = \{4\}\\
&G \cup \{s_{K+2}\} = \{1,2,3,6,7\}
\end{aligned}$$

Table 5: Sample database (Example III)

S	DK(S)
1	4
2	2
3	1
4	8
5	9
6	6
7	5

The standard procedure requires forming queries from the combination of H_0 and every combination of p elements in $G\cup\{s_{k+2}\}$. However, this is no longer possible because H_0 contains one element and $p = 3$. In this case, our queries would have to be of size 4 which cannot be done. Instead, we form every query that can be made from the combination of H_0 and every combination of required elements to make queries of size k. We get the following queries and responses:

f(*DK*(1), *DK*(2), *DK*(3), *DK*(4), *DK*(6)) = *f*(4, 2, 1, 8, 6) = *f*(1, 2, 4, 6, 8) = 6
f(*DK*(1), *DK*(2), *DK*(3), *DK*(4), *DK*(7)) = *f*(4, 2, 1, 8, 5) = *f*(1, 2, 4, 5, 8) = 5
f(*DK*(1), *DK*(2), *DK*(4), *DK*(6), *DK*(7)) = *f*(4, 2, 8, 6, 5) = *f*(2, 4, ~~5~~, 6, 8) = 6
f(*DK*(1), *DK*(3), *DK*(4), *DK*(6), *DK*(7)) = *f*(4, 1, 8, 6, 5) = *f*(1, 4, ~~5~~, 6, 8) = 6
f(*DK*(2), *DK*(3), *DK*(4), *DK*(6), *DK*(7)) = *f*(2, 1, 8, 6, 5) = *f*(1, 2, ~~5~~, 6, 8) = 6

The standard procedure requires a median to occur p times and a second median to occur only once. We do not have a median that occurs p times because we got more than p+1 queries. However, we still managed to get two unique medians. One that occurred more than p times and another that occurred only once. We apply the same conclusion and obtain:

$$DK(6) = 6$$

Despite the differences, we compromised an element by following the same procedure with a few modifications. These modifications showed successful compromises for many of our different tests. The modifications are formally described as follows:

Modification #1: We will make a sorted list of the medians obtained from our first k+1 queries. Because k is odd, we will obtain an even number of medians. We find the lower and higher value median from the list of medians. If they are different, we say that g is less than or equal to the lower median and h is greater than or equal to the higher median. If both medians are equal, we take the following approach. We make a sorted list of the unique medians (remove repetitions) and repeat the previous process in this new list. If the length of the unique median list is odd, we say that h is greater than or equal to the median of the list and g is less than the median of the list. The following examples summarizes the different possibilities for query sizes of $k = 5$:

Medians: 1, 1, 2, 3, 4, 5 → $g \le 2$, $h \ge 3$
Medians: 1, 2, 2, 2, 4, 5→ Unique Medians: 1, 2, 4, 5 → $g \le 2$, $h \ge 4$
Medians: 2, 2, 2, 2, 4, 5 → Unique Medians: 2, 4, 5 → $g < 4$, $h \ge 4$

Modification #2: Due to randomization, it is possible for the query made from the elements in $G'\cup H\cup\{s_{k+2}\}$ to yield a response less than the value of h. If this happens, we apply the same observation as if we had got the value of h, i.e., $DK(s_{k+2}) < h$.

Modification #3: For the last queries, we may be able to make more or fewer than p+1 queries because we obtained different sizes for the sets G and H. We will combine the set of H_0 with every combination of x elements of $G\cup\{s_{K+2}\}$, where $x = k -$ size of H_0.

Modification #4: Because we may obtain more or fewer than p+1 queries in the end, we change the number of occurrences of one median to be greater than or equal to 2. The second median is still required to occur once.

Note the modifications implemented do not affect the standard procedure. One can apply the inference procedure with the modifications described above to a database without randomization and obtain the exact same results. At this point, let us define what will be considered as a fail procedure. A fail procedure can occur if and only if one of the following conditions is present:

- The initial k+1 queries yield the same median, i.e., we obtain one unique median
- We obtain one or more than two unique medians from the last queries, or
- We do not obtain one median with one occurrence and another median with the remaining occurrences

As stated earlier, a successful procedure is not necessarily a successful compromise. The inference attacker cannot tell if the successful procedure is a successful compromise. In other words, the obtained result from the inference procedure can be incorrect (as shown previously). In the next section, we explain our simulation approach and the analysis of our results.

Results and Discussion

In this section, we discuss the simulation and analyze it. Also, we point out the error magnitude and accuracy level. Furthermore, we describe how to implement randomization on an existing database.

Simulation Results and Analysis

For our research, we implemented a simulation to simulate statistical queries of type median using our proposed randomization technique. The queries requested are made with the intention to compromise database elements using the modified inference procedure. The entire simulation was done under the assumption that all elements are pairwise distinct. This is certainly the assumption in the theorem that is the basis of the simulation. For each of the different parameters, the simulation was run a million times on randomly generated databases. We set our database size to be N = 500 for all iterations with distinct values ranging from 0 to 999. Every 10 iterations, a new database with new distinct values is randomly generated. For our test cases, we use of the following values for k:

5, 15, 25, 45, 95

For each value of k, we have a simulation for the following values of the tolerance number t:

1, 2, 5, 10, 20, 50

To show our results, we will be looking at two different sets of graphs. The first set summarizes the obtained results from the inference procedures done in each of the simulations. The second set summarizes the frequencies of type of values returned as a query response, i.e., frequency of i, j, m, n, or p responses. Each graph summarizes all the results from all simulations done on a value of k. The following 5 graphs (Fig. 2-6) show the inference procedure results.

By looking at the charts, we can make the following observations:

- As the value of t increases, so does the fail procedure percentage
- As the value of k increases, the percentage of successful compromises increases slightly and the percentage of incorrect results increases slightly
- As the value of k increases, the percentage of fail procedures decreases

The observations were as expected. The third observation correlates with the example given earlier of the sample database made up of consecutive values. In that example, we showed that the percentage of a successful compromise was 16.67% and the percentage of fail procedures was 66.67% which seems to be the direction of the percentage as the value of k gets closer to the value of N.

The following graphs (Fig. 7-11) show the response frequencies for the simulated queries.

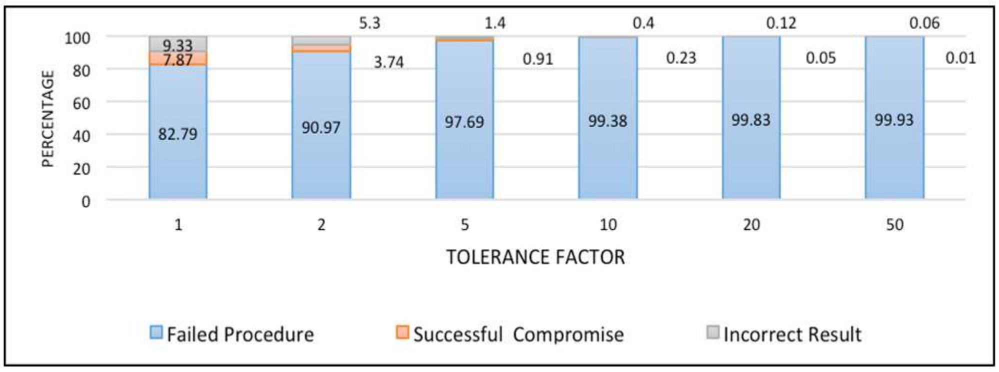

Fig. 2: Inference procedures k = 5

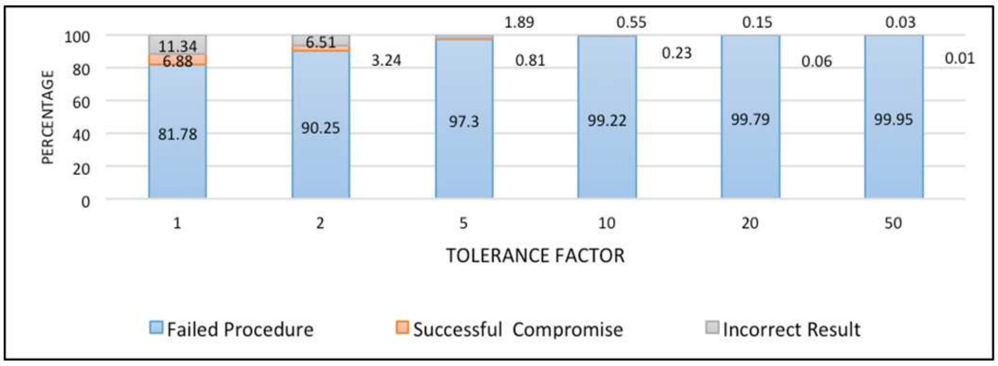

Fig. 3: Inference procedures k = 15

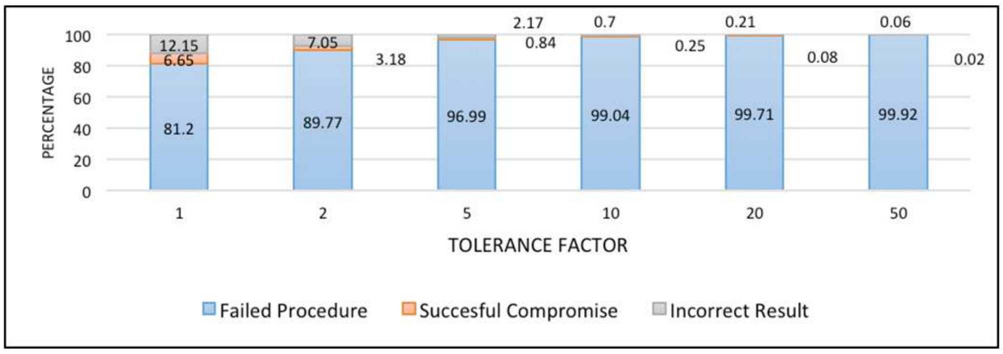

Fig. 4: Inference Procedures $k = 25$

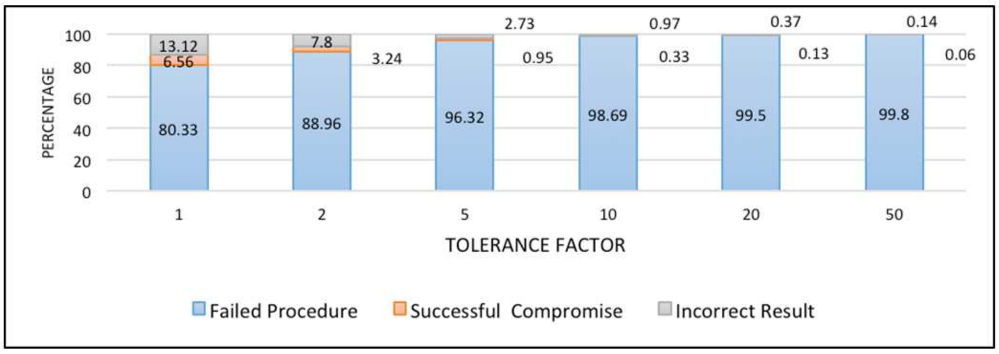

Fig. 5: Inference procedures $k = 45$

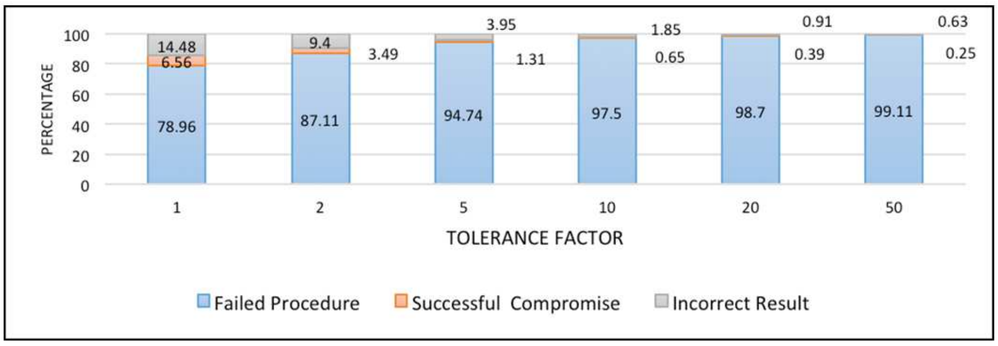

Fig. 6: Inference procedures $k = 95$

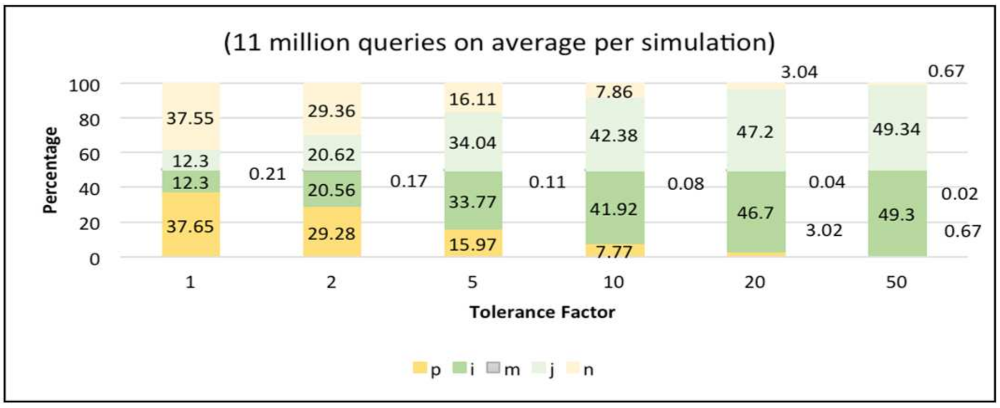

Fig. 7: Query response frequencies $k = 5$

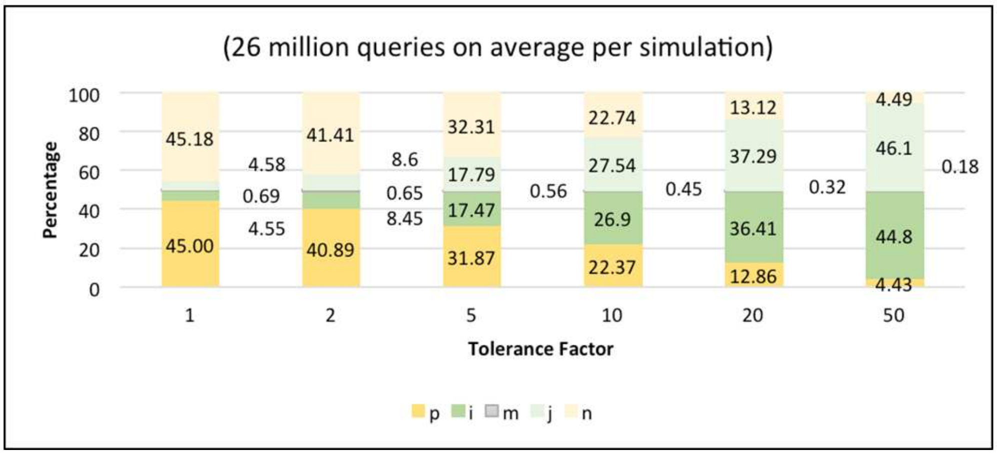

Fig. 8: Query response frequencies $k = 15$

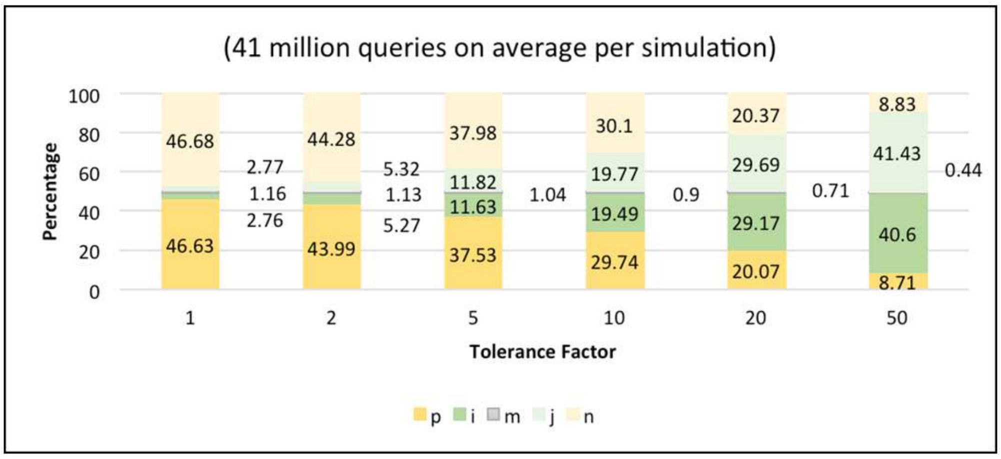

Fig. 9: Query response frequencies $k = 25$

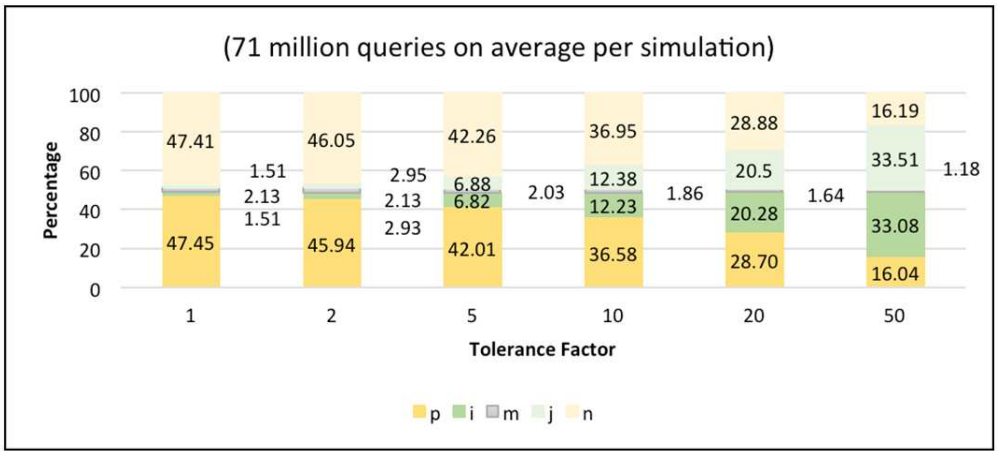

Fig. 10: Query response frequencies $k = 45$

There are three observations we can make here:

- As the value of t increases, so do the frequencies of i and j responses
- As the value of k increases, so does the frequency of m responses

As the value of k increases, a higher value of t is needed to increase the frequency of i and j responses

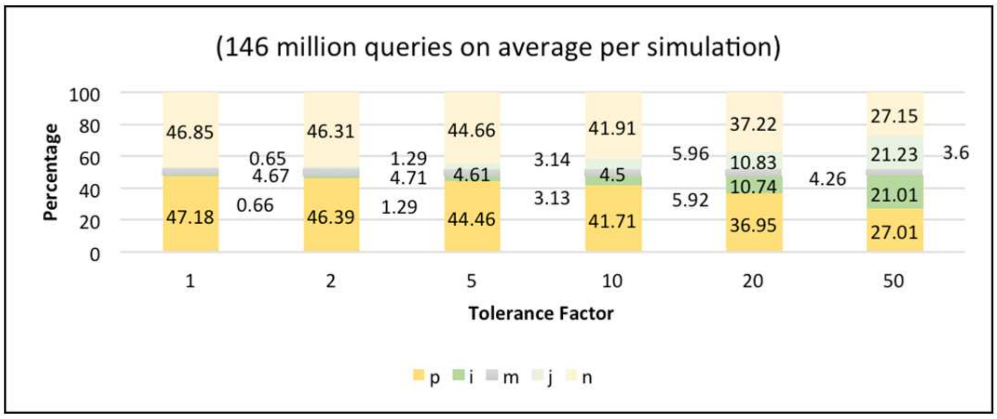

Fig. 11: Query response frequencies $k = 95$

The observations were as expected as well. As the tolerance increases, it is more probable to find an *i* or *j* value. Similarly, as the query size increases, we are less likely to get randomly elements with an *i* or *j* value (because there are fewer elements in the database to choose from). Therefore, the frequency of responses of type m increases. By the same token, the tolerance number may not need to be very large for query sizes that are relatively close to the size of the database. Since there are fewer values to choose from, one needs fewer tries to find such a value.

Based on these observations and the ones made on the inference procedure results, we came up with the following conclusions:

- *i* or *j* responses decrease the rate of successful compromises
- The higher the tolerance, the more likely one is to find an *i* or *j* value
- Higher tolerances are needed as query size increases (although this depends on the size of *N* as well)

Using these conclusions, we can determine an appropriate tolerance for some specific database. One can see from the results that with a tolerance of 5, randomization stops about 97% of the inference procedures. More importantly, it is impossible for the attacker to know whether a correct or incorrect result was obtained from the remaining procedures.

It is important to analyze the validity of the simulation parameters in real databases. In this simulation, we have sampled queries that are completely random. In reality, this may not be the case. Requestors may want statistical information on certain indices that just happen to lie in a subset of the entire database. Imagine asking for the medians of certain queries that are comprised entirely of indices whose responses, once sorted, would lie in the middle 20% of the database. In such scenario, the other 80% of the database elements would not yield any contribution to randomization because those elements cannot be *i* or *j* values. If this were to occur (whether it is intended or not), it would mimic a smaller database. That is, the value of *k* would be closer to *N*. As described from our observations, a larger tolerance number would benefit these cases. Nonetheless, randomization still protects databases under these conditions due to the possibility of incorrect results and the uncertainty of the inference procedure. However, it is still recommended for the database managers to pick adequate tolerance number depending on the type of information being served by the database and its performance requirements.

Errors and Other Remarks

Earlier, we briefly discussed the error introduced. At worst, the error in the query response is the previous or next element. This error can be very big or very small depending on the type of data. However, relative to the given query, the error is relatively small. By the same token, the error obtained from the inference procedure on randomized queries is also small. In other words, the results obtained by the attacker are relatively close to the real result (and in some cases, as shown, the actual result). It is the number of fail procedures and level of uncertainty what makes randomization a secure approach.

Throughout our tests, we ran different implementations that were not useful for queries of type median. The removal of the median from every query was shown earlier to fail against some procedure modifications. The only other method that showed favorable results was a similar approach to the one we suggest. In fact, it was tested before and gave better results than the current suggested technique. However, it was not a viable option for most databases as it required to search the entire database on every query. The method consisted of using the gaps to find the value closest to the median in the specific range. The results were more consistent throughout queries, which gave more incorrect results to the attacker.

One important thing to note is that we tested this technique on databases composed of distinct values.

Databases may contain repeated values. If the value of *p* is equal to the value of m, one could change the value of *p* to be the previous different value element in the sorted query. Hence, a gap could still be used. However, this will increase the relative error since one is moving further away from the median, relative to the query. Although we believe randomization does provide security for databases including repetitions, we did not test such a claim. Our implementation depends on the tolerance number and relative error and is subjective to the number of repetitions present.

Implementation of Randomization to Existing Databases

To determine the appropriate value to use as part of the randomization technique, we had to find the median of the query. The steps followed up to that point are the same steps followed as if randomization was not used. The associated query values must be found and sorted. Then, the median value is found and returned. With randomization, one follows the exact same process but adds the randomization step after finding the median value. In other words, implementing randomization into an existing database is relatively simple. Before the median is returned, the analysis of the gaps is made and an attempt to retrieve an element fitting in the appropriate gap based on the appropriate case is made (up to as many times as the selected tolerance). If the value is found, it is returned. Otherwise, the true median already found is returned. With randomization, the extra calculations are finding the *I* and *J* magnitudes, requesting a random element from the database and checking if the value is in the range involving the largest gap. The last two steps are repeated as many times as the manager needs to (determined by analyzing the trade off between security and performance).

Conclusion

Randomization is a technique simple to implement on existing databases. It can protect private information from statistical queries of type median by ensuring low probabilities of successful inference attacks. Despite being possible to attain a correct inference, it is impossible for the attacker to be certain of the inference results because of the uncertainty obtained during the inference procedure. The simulation results are favorable and demonstrate the security it offers. As shown, it is still dependent on the database manager to choose the correct parameters to meet the needs of the database. In conclusion, we see randomization as a viable option for statistical databases that need to provide statistical information and are limited by the current inference control techniques.

Acknowledgment

Support under NSF grants IIS-1359199 (REU Site Program) and DGE-1433817 (SfS program) is acknowledged.

Author's Contributions

Jose Daniel Velazco: Made considerable contributions to this research by running the simulation and analyzing the data; he also contributed to drafting and critically reviewing the manuscript for significant intellectual content.

Mohammed Awad: Made considerable contributions to this research by interpreting the data; he also contributed in critically reviewing the manuscript for significant intellectual content.

Ernst L. Leiss: Designed the research plan and organized the study; he contributed to the presentation and analysis of the results; he made a critical review of significant intellectual content and also added genuine content where applicable.

Ethics

The authors confirm that the article is original, contains unpublished material, and that there are no ethical issues associated with its publication.

References

Adam, N.R. and J.C. Worthmann, 1989. Security-control methods for statistical databases: A comparative study. ACM Comput. Surveys, 21: 515-556.

Cavallo, B. and G. Canfora, 2016. A probabilistic approach for disclosure risk assessment in statistical databases. Quality Quantity, 50: 729-749.

Cynthia, D., 2006. Differential privacy. Proceedings of the International Colloquium on Automata, Languages and Programming, (ALP' 06), pp: 1-12. DOI: 10.1007/11787006_1

Garcia, J., J. Wilder and E.L. Leiss, 2010. Inference control in statistical databases: An improved randomization approach. Proceedings of the Conferencia Latinoamericana de Informatica, Oct. 18-22, Asuncion, Paraguay.

Giggins, H. and L. Brankovic, 2012. VICUS - A noise addition technique for categorical data. Proceedings of the Data Mining and Analytics, (DMA' 12), Sydney, Australia, pp: 139-148.

Heatherly, R., M. Kantarcioglu and B. Thuraisingham, 2013. Preventing private information inference attacks on social networks. IEEE Trans. Knowl. Data Eng., 25: 1849-1862.

Hegadi, R.S., I.M. Umesh, T.N. Manjunath and G.K. Ravikumar, 2011. A survey on recent trends, process and development in data masking for testing. Int. J. Comput. Sci., 8: 535-544.

Leiss, E.L., 1981. On the security of randomized databases: A simulation. Tech. Rep. #UH-CS-81-01, Department of Computer Science, University of Houston, Houston Texas.

Leiss, E.L., 1982. Principles of Data Security. 1st Edn., Plenum Press, New York.

Leiss, E.L., 1982. Randomizing, a practical method for protecting statistical databases against compromise. Proceedings of the 8th International Conference on Very Large Data Bases, Sept. 8-10, Mexico, pp: 189-196.

Lu, H., J. Vaidya, V. Atluri and Y. Li, 2015. Statistical database auditing without query denial threat. (n.d.). Informs J. Comput., 27: 20-34.

Muralidhar, K. and R. Sarathy, 1999. Security of random data perturbation methods. ACM Trans. Database Syst., 24: 487-493. DOI: 10.1023/A:1025610705286

Muralidhar, K. and R. Sarathy, 2003. A theoretical basis for perturbation methods. Statist. Comput., 13: 329-335. DOI: 10.1023/A:1025610705286

Naveed, M., S. Kamara and C.V. Wright, 2015. Inference attacks on property-preserving encrypted databases. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS' 15), ACM, New York, pp: 644-655.

Sarathy, R. and K. Muralidhar, 2002. The security of confidential numerical data in databases. Inform. Syst. Res., 13: 389-403. DOI: 10.1287/isre.13.4.389.74

Zou, G. and M. Zhang, 2012. A new data perturbation method of reference control in statistical database. Applied Mechan. Mater., 3134: 241-244.