Final Report

Aaron Heckman - U02266356
Xiankang Wu - U59844514

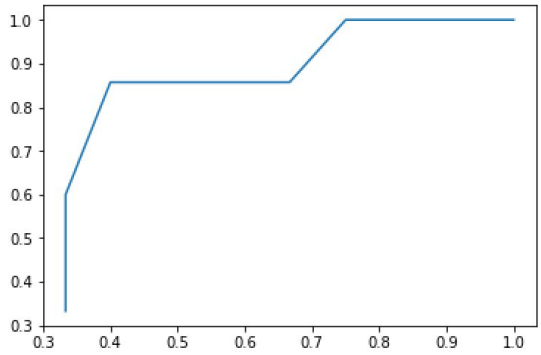
Project overview:

In this project, initially we wanted to implement a tracing attack so that we could learn more about them, and then use linear regression prediction models to expand the dimensionality of the data set in order to make tracing attacks work better on a smaller array of data.
We found that linear regression was a bit too complex for a project of our scale, and so we scaled it back to focus more on comparing Randomized Response and Laplace noise using estimation and resistance to tracing attacks as a criteria.

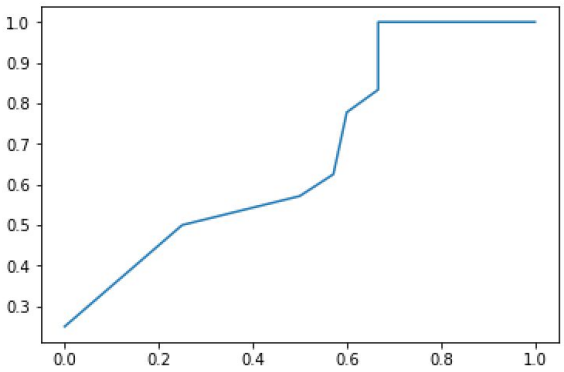
Project description:

There are several steps for our project, so in this section we are going to discuss what we managed to achieve.
First, we implemented a very basic version of tracing attacks. We created a database of randomly generated bits based on a bit-flipping mechanism, which we then perturbed using Randomized Response. Then, our tracing attack generates random guesses and tries to determine if they are in the table or not. To visualize this, we generated several ROC curves using the same parameters for data. Here are two ROC curves generated with a database that has 1000 rows, 10 columns, and a bit-flipping probability of .2, with 10 guesses per database and 10 databases generated.

```
In [479]: roc_generator(1000, 10, .2, 10, 10)
```




```
In [480]: roc_generator(1000, 10, .2, 10, 10)
```



A quick note on ROC curves: they are obtained by finding the true positive rate of a guess and graphing it in terms of the false positive rate. The Y-axis represents the true-positive rate, while the X-axis represents false-positive rate. The steeper an ROC curve looks, like the one on the left, the better the attacker is at guessing correctly.

As we can see, the results are a bit random. The curve on the left shows a good guess rate, but the one on the right indicates a guess that is much worse. This comes from the weak nature of our attacker, as well as the fact that we had trouble testing on very easy or very hard parameters, because they would cause our true positive rate or false positive rate to become zero. We decided that the weak nature of our attacker was something we could fix in the next step, and we understood enough to move on.

Note: After this step, we tried to find scientific literature on how to use linear regression to predict how a data set was organized and use that to expand our database to be more descriptive, and thus, more vulnerable to tracing attacks. After discussing this with the professor, we decided that this route was too complicated for our project. He also gave suggestions for how we could take our experiment in different directions, so we did.

Secondly, we implemented some of the professor's ideas, which were the following:

- Instead of a linear query database, we changed our mechanism to release a frequency of each of the k attributes in our dataset. That is, for each column in the database, we calculate the frequency of the appearance of 1s. Here is an example where k = 10. In our Laplace mechanism this is the only thing the attacker sees, but in the RR mechanism, they also get to see a table that has been perturbed..

```
freqs =  [473, 475, 481, 490, 502, 481, 509, 461, 509, 492]
```

- We implemented a second method of perturbing the data using Laplace noise. We generate a number from a Laplace distribution with theta set so that the accuracy of the noise table is equal to the accuracy of our Randomized Response table, and then we add that number to a frequency, and repeat the process across all frequencies in the table.
- Instead of our attacker generating his guess randomly, we altered our code so that the attacker starts with a row, representing someone's data which may or may not be in the data set, and the attacker has to determine if the data they got is in the dataset or not. Here is an example for a 100x10 table, with bit-flipping probability e = .2, averaged over 10 different guesses. The output here is RR true positive rate(tpr), RR false positive rate(fpr), Laplace tpr, and Laplace fpr, respectively.

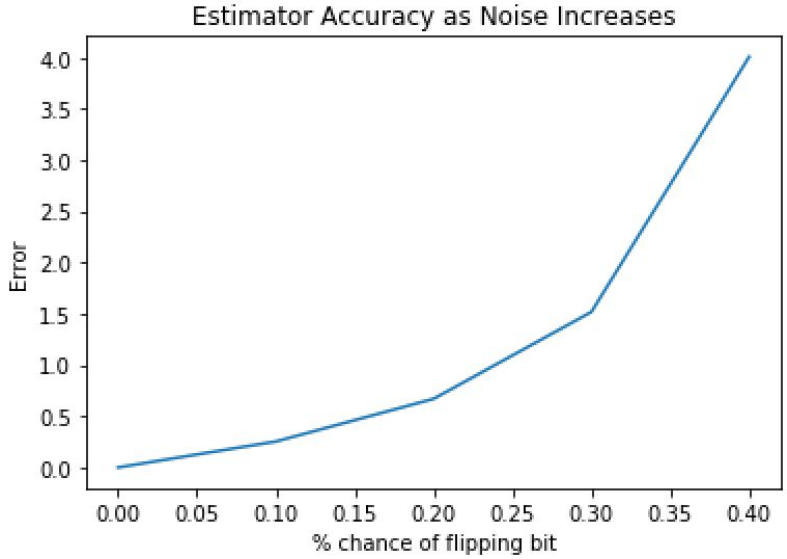
```
In [1693]: tracing_compare(100, 10, .2, 10)
Out[1693]: [0.75, 0.6666666666666666, 0.75, 0.8333333333333334]
```

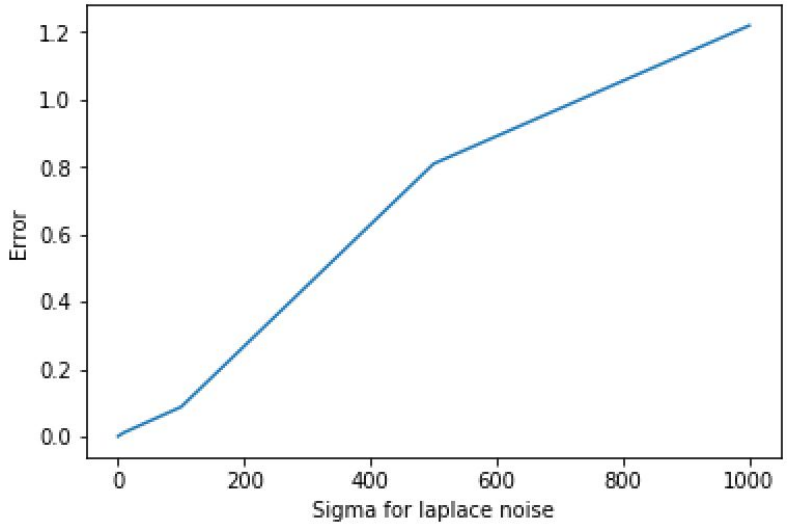
- The way that we equalize the accuracy is by calculating an estimation for each mechanism at a given e. We increment the sigma for the Laplace noise estimation until it is roughly equal to the RR estimation

Link to code:
https://github.com/JerryWu96/cs591-final-project

Results and analysis:

Our significant results come in two flavors: estimation and tracing attack resistance.

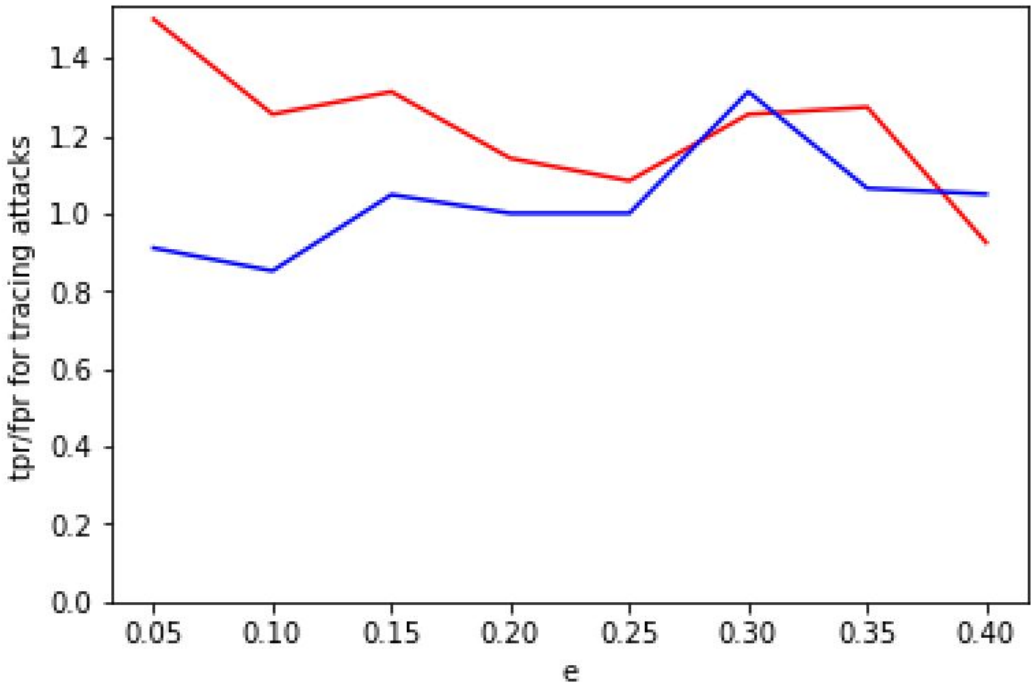




This is a plot for our estimator on a 1000 by 20 table. They represent degree of error as noise increases. In both graphs, the y axis is error as a function of estimated sum/actual sum. So if the actual sum of a column is 565 and the error on this graph is 1, that means that the estimation function guessed either 1130 or 0, since those numbers are 565 away from the actual sum. The x axis is labelled for each graph.

We can see from this graph that you need a very large sigma to reach inaccuracy levels comparable to those seen in a small increase to e. Anything past 1 is usually a pretty useless estimation. On their own, this data doesn't tell us much, but we use it to inform our calculations for the next, and far more interesting stat.

This is a plot of tracing attack performance over e. The y-axis is tpr/fpr, which means that the higher the point, the better the tracing attack performs at that level of accuracy. Red line indicates attacks on RR systems, and blue line indicates attacks on Laplace systems.



From this graph, we can see that Randomized Response generally protects data worse than Laplace noise. Laplace's line is lower, meaning that for the same accuracy its tracing attack performs worse. There is a general downward trend in RR that isn't really present in Laplace. This downward trend is exactly what we would expect in RR, the less the noisy table lines up with the actual one, the harder it will be to guess. There are a couple of reasons the trend could not be present in the Laplace mechanism. First, and much less likely, adding Laplace noise guarantees a degree of privacy. It may be that the Laplace noise provides a degree of privacy at a certain noise level and we have far exceeded that noise level, rendering any increases meaningless. Secondly, and much more likely, is that our tracing attack is faulty and this is the general behavior of it. The attack, which you can find as tracing_compare() in our code, establishes a threshold based off of the inner product of the data the attacker starts with and the noisy sums. It is possible that the way that threshold is found is faulty, or that even the idea of finding a threshold and sorting the guesses that way is bad innately.

Potential Improvements:

- Verifying that our Laplace attacker does something reasonable is imperative. If our attacker is doing the best thing that he can, then we are performing an accurate test of the resistance of our mechanism to tracing attacks. If it is not, then we need to revise our experiment and the results become much less significant.
- Our code could be optimized in a number of ways, both in terms of speed and performance. Namely, we can't run it on obvious problems, like where n >> k to the point that every possible string of bits is in the table. This is because we use the results of the test to find tpr and fpr, and these include a division and this occasionally includes a division by 0. Error handling like that is very much needed in the code.
    - On a similar note, our estimation function doesn't work when e = .5, because the RR estimator uses the equation (column_sum[i] + n*e)/(1-2*e), and 1-2*e = 0 when e = .5.
- The way we equalize sigma and e is very rudimentary. We simply have a list of sigmas that we go through until we get an estimation that is more than the estimation for e and we assume that that is close enough. This means that we either need to add in a lot of granularity, or revise the way we equalize e and sigma.