CHORUS is Porous

Attacking Implementations of Differential Privacy

Amaresh Ankit Siva

April 25, 2020

Contents

Abstract

In this work I explore the vulnerability of CHORUS[8] and FLEX[9], using a sidechannel attack [12]. CHORUS and FLEX are differentially private querying mechanisms jointly worked on by Uber and UC Berkeley. They aim to provide noisy results to an analyst's queries. I propose and implement an algorithm to use the exploit, and analyze its efficacy. Finally, I implement the proposed defense and verify that it protects against this exploit.

Keywords differential privacy, querying mechanism, sidechannel attack

1 Introduction

Differential privacy is a tool and a new area of research that seeks to balance the economic incentives of machine learning with the ethical considerations surrounding using personal data to that end[3]. It seeks to prevent violations of privacy that may stem from unexpected instances. For example, it has been found that a word prediction algorithm leaked social security numbers that happened to be in the training set [1]. Differential privacy provides a strong mathematical definition for the word privacy [6]. It promises to the individual whose data is contained in a database that they are no more likely to be affected by the outcome of the analysis than if they were not in the database [5]. With the new restrictions on data consumption coming into the world stage such as the European Union's General Data Privacy Regulation and the California Consumer Privacy Act, differential privacy has the potential to very practically impact the enterprise world [3]. Specifically, by providing a quantitative approach to the usually qualitative ideation of privacy.

It is in this vein that Dawn Song et al. from the University of California, Berkeley collaborated with a team from Uber to publish the paper "Towards Practical Differential Privacy" [8]. It takes a novel approach to differential privacy, with its ideation of SQL (Structured Query Language) re-writing as a possible mechanism to introduce noise to queries. Unfortunately, for all its proposed innovations, the promise of differential privacy has eluded this team. A paper published by Ilya Mironov five years before "Towards Practical Differential Privacy" made note of an important implementation flaw that could occur [12]. Namely that there was no consideration of the rounding of floating point arithmetic when the mathematically proven algorithms were implemented on real-world systems [12]. The details of the attack itself will be detailed in the following sections but it is important to note at this point that there were other concerns with the implementation raised by Frank McSherry [10][11]. The concerns raised by McSherry were mainly regarding Song et. al's handling of joins. A join is a database operation where the rows of two input tables are matched by equality of some field present in both and concatenated. This is a difficult operation to support since joins could potentially change the sensitivity of the query being computed drastically. The sensitivity of a query is important because it is a factor of the scale of noise we add to the output of the function, and thus the privacy we can ensure to the user. In this work, I will focus on only reconstructing one table and will not be using joins in my attack. Likewise,

my implementation of a fix will only resolve the arithmetic flaw, not the conceptual issues with handling joins.

The implementation gap in Song et al's work can be exploited using the attack described by Mironov. However, proving that it fails to be differentially private is not sufficient. It remains to be shown that plugging the leaks with the *snapping mechanism* [12] prevents the exploits described in the Mironov paper.

Thus in this thesis, Mironov's attack is implemented against FLEX and CHORUS. The original code is also augmented using a known security patch to successfully defend against said attack. First, I will discuss relevant literature in section 2. The details of the algorithm constituting the attack are laid out in section 3. Finally, results and discussion are presented in 4 and 5 respectively.

2 Literature Review

2.1 Background

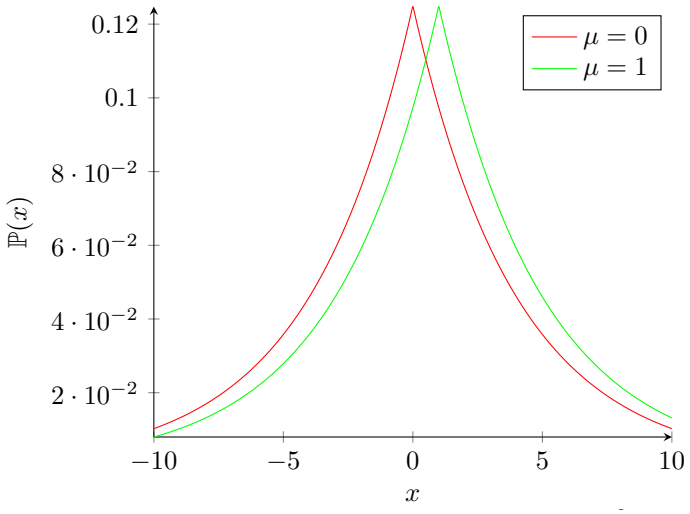
Differential privacy is a mathematical definition of privacy that provides a strong worst-case guarantee against additional individual harm. The definition also provides for a way to estimate how private a certain algorithm or analysis performed on a data set is. It was introduced by Dwork and McSherry [6] in 2005 and many algorithms for it have been developed since. The crux of this approach is to bound the likelihood of an adverse outcome happening as a result of an individual being present in a dataset. More formally, it is represented as:

$$\mathbb{P}(f(x) \in S) \leq e^{\epsilon}\mathbb{P}(f(x') \in S) + \delta \tag{1}$$

Where x and x' are neighboring databases. Neighboring databases are databases with one row changed, because of the addition, deletion, or edit of a row of the database. We can consider this in the context of an Uber database. x can be understood as a database in a universe where Alice was an Uber user and x' as a database in a universe where she was not. f can be considered a function or analysis that an analyst at Uber is trying to run on the database. To explain the definition, assume that the analyst is counting the number of users with the name Alice with a differentially private algorithm and she is the only one with her name. S is the set of outcomes of this probabilistic function. What the equation says is that the adverse outcome's likelihoods of occurring are bounded by ϵ and a failure of privacy factor δ. For a given ϵ and δ, an algorithm that satisfies equation 2 is said to be (ϵ, δ)-DP. That is to say that in figure 1, the probability of the count returning 1 in the universe with Alice in the database is at most a factor of ϵ more than the probability of the count returning 1 in the universe where she is not in the database. Figure 1 uses noise sampled from a Laplacian distribution to "muddy" up the counts.

The Laplacian distribution is frequently used because of its long-tailed nature and nice composition properties. It also allows for the use of $(\epsilon, 0)$ differential privacy (or just referred

Figure 1: Two Laplacian distributions



to ϵ-DP) because its tails decay at infinity. The probability density function is defined as:

$$f(x) = \frac{1}{2\lambda} \exp(-\frac{|x - \mu|}{\lambda}) \tag{2}$$

The parameter λ in the function above is $\frac{\delta f}{\epsilon}$. So to achieve $\epsilon = 1$ differential privacy for a query that has sensitivity 1, λ would be set to 1.

Differential privacy relies on adding noising to some step so as to reduce the accuracy.

A differential privacy algorithm has two tasks set out for it:

1. Finding the sensitivity of the data set. That is, identifying how much the value of the analysis can vary over all possible entries of the data set. In our example, the sensitivity of the count is 1 since only one Alice exists in the universe.

2. Generating noise sampled from a Laplacian distribution parametrized by the ratio of the sensitivity found in the previous task and the ϵ (epsilon) value decided by the analyst.

The ϵ is the essence of the privacy measurement and usually, a value between 0 and 1 is considered a good guarantee of privacy [6]. Large institutions like Apple, Google, and the Census Bureau have begun introducing differential privacy to their data science pipelines.

2.2 Related Work

An attack proposed by Dinur and Nissim [4] tested by Cohen and Nissim [2] on a privacy mechanism that sits outside of differential privacy. The mechanism in question, Diffix, is the product of Aircloak. It uses noise sampled from a Gaussian (also known as a Normal) distribution to retain individual privacy [2]. The key difference between Diffix and Laplace

distribution based solutions is that Gaussian noise has zero support for outcomes of neighboring databases. Nissim and Cohen create a linear reconstruction attack by modeling the attack as a linear program (LP). LPs are a well-known problem in computer science and have many computationally efficient algorithms devoted to solving them. Thus Cohen and Nissim found a computationally efficient way to recreate the database. However, this cannot be readily translated into either the FLEX system or the CHORUS as it is not certain if discontinuous Laplace noise can be attacked by a linear program.

2.3 Differentially Private Querying Mechanisms

Recently, progress has been made in efficiently computing smoothed local sensitivity in novel ways. A contribution in this direction has been made by Song, Johnson and Near [9]. They have developed "Elastic Sensitivity" which is a computationally efficient method of finding the sensitivity of a query being run on a database. To satisfy the second task they introduce FLEX and a novel query rewriting solution called CHORUS. This paper was the result of a collaboration between the University of California Berkeley and Uber. The latter of which provided the authors of the paper access to the queries being run by their data analysts, to better tailor the CHORUS and the FLEX to real-world systems [9]. The authors also produce a list of statistics on the types of queries run within Uber, thus allowing for both solutions to be compatible with any type of database and support querying across many tables within the same database. The CHORUS works by taking the analyst submitted query and transforming it by adding an appropriately scaled noise term. The FLEX system is simpler in that it only adds the scaled noise term to the output of the query (rather than changing the query and doing nothing to the output). CHORUS is an ensemble of methods, including FLEX. An important question to answer is whether the output of either system (CHORUS or FLEX) adheres to the definition of differential privacy [6]. Interestingly, this question is raised by a paper published 5 years prior.

The paper, published by Mironov in 2012 [12] describes an attack on several differential privacy solutions at the time. Examples include PINQ and Airavat. The attack involved looking closely at the noise generation mechanism within the second task. Mironov identified a key weakness in programming languages and the way they distribute the real numbers within the confines of the discrete space they are limited to. Doubles are the most often used type of data that are the best approximation of the real numbers. The standard that these programming languages follow, IEEE 754 [7] has been around since 1985 and is the most widely used standard. It represents numbers with 64 bits. 52 bits represent the mantissa, 11 bits represent the exponent and the remaining bit is used to represent the sign of the number. A property of this standard is that there exists a difference in the density of representation of real numbers in the standard. That is, there are 2^{52} doubles between 0.5 and 1, which is the same as the number of doubles between 0.25 and 0.5 (and between 0.125 and 0.25, etc.). Usually, in non-critical applications, the accrued floating point errors that result from this flaw are small enough to be ignored.

$$L \to \frac{x}{|x|} \cdot \lambda \ln(1 - 2|x|) \text{ given that } x = U - 0.5 \tag{3}$$

The above equation transforms a uniform random variable(U) to a Laplacian random variable(L). The floating point errors in applying the ln() (natural log) function to these doubles leads to a discontinuous output. This function is applied to doubles every time noise is sampled from a Laplacian distribution, hence the floating point errors matter a lot.

However, within differential privacy, it is a requirement from the definition that the outcomes of two neighboring databases be equally likely or a small multiplicative factor away from one another. The fact that not all doubles between 0.25 and 0.5 are represented by the output $ln()$ leads to the result that it is possible to discern one neighboring database from another, at least 40% of the time. This is a glaring violation of the definition itself. While Mironov sets out a partial description of the attack, it does not cover the potency of the exploit in a query rewriting system. Mironov provides a defense against his attack, but like the efficacy of the attack on the codebase at hand, it remains to be shown as an effective defense.

This study hopes to successfully or unsuccessfully attack and defend the codebase attached to the paper by Song et al. [6]. Understanding why a specific attack or defense works or does not will also help with realizing differential privacy in practical scenarios and prevent abuse of mechanisms deployed to the real world. Especially considering the fact that the solution is being made larger without paying attention to this core flaw. The attack will likely require the exploit outlined by Mironov [12]. It will also remain to be seen if the resulting attack follows the same linear program property introduced by Nissim and Cohen[2]. Both FLEX and the CHORUS are built with languages that use the IEEE 754 [7] standard and the noise generating mechanism has already been shown to be vulnerable to the neighbor discerning attack. I will write an implementation attacking both mechanisms (specifically using elastic sensitivity) and a defense against it.

3 Methodology

In this section, I will describe CHORUS and FLEX in 3.1 as a black box and the attack on black boxes in 3.2.

3.1 CHORUS and FLEX as a Black Box

Both CHORUS and FLEX work to add noise to counting queries. They are both black box approaches that promise to return differentially private results from a database for counting queries provided by an analyst at a given privacy budget. Figure 2 depicts this. The querying mechanism stands as a barrier between the analyst and the table. It may or may not mimic the original interface of the database but performs both tasks of computing the sensitivity and adding noise to the result.

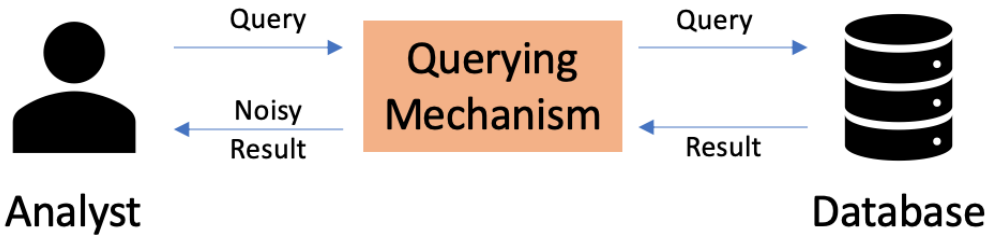


Figure 2: A general black box DP querying mechanism

FLEX very closely mimics the black box as can be seen in figure 3. There is an additional advantage to FLEX: that the analyst continues using the same querying language as they did before. FLEX intercepts queries, computes the sensitivity and required noise before releasing the summed noise and original result to the analyst.

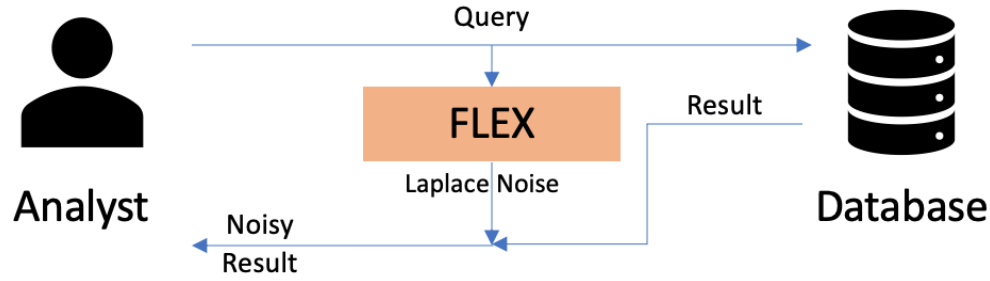


Figure 3: An overview of how FLEX works

CHORUS differs in approach by intercepting the query as FLEX did, but instead it computes the sensitivity of the query and rewrites the query to make the database management system (that handles computing any query on the table) compute the noise and add it to the result. It does not intercept the result and provides the same advantage of not requiring the analyst to relearn a querying language.

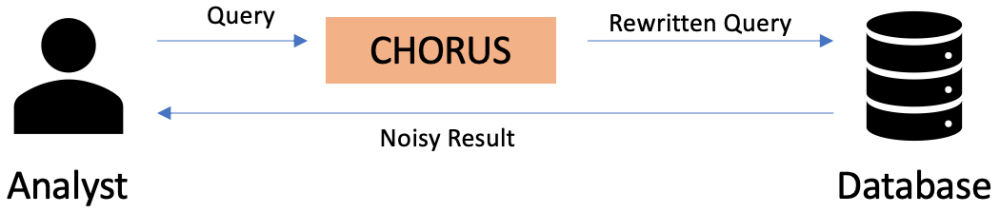


Figure 4: An overview of how CHORUS works

For example, CHORUS would rewrite the MySQL query below...

```
SELECT COUNT(*) FROM user_table
WHERE username LIKE "A%";
```

... to the following query:

```
SELECT COUNT(*) + 2000000.0 * (CASE WHEN RAND() - 0.5 < 0 THEN -1.0 ELSE 1.0 END\\
* LN(1 - 2 * ABS(RAND() - 0.5)))
FROM (SELECT username FROM user_table) WHERE username LIKE "A%";
```

Both FLEX and CHORUS use the same formula 3 to source the Laplace noise, thus making both vulnerable to Mironov's attack. Hence, my attack algorithm and implementation are on a black box that could either hold either the FLEX or CHORUS methods. The results are identical for both since their noise sampling is too. I will hereafter refer to the mechanism protecting the database plainly as a querying mechanism.

3.2 Attack on a Black Box

The assumptions held about the attacker in this case are that they know:

1. the per-query epsilon value
2. the approximate number of records
3. the format of the table and metadata

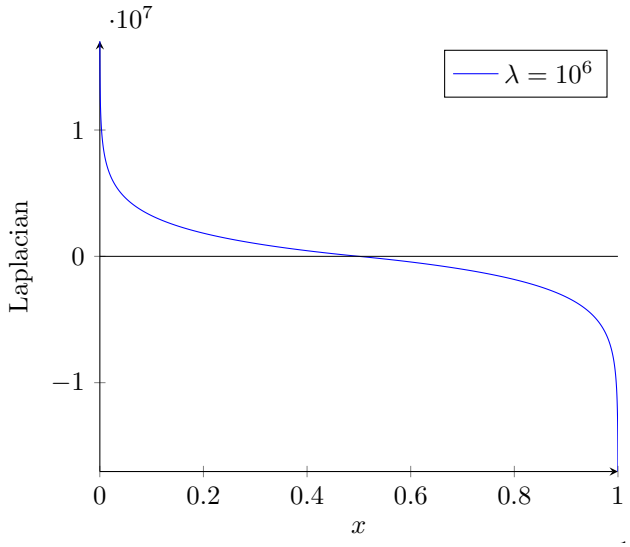
To attack a database protected by a querying mechanism, I primarily use the three procedures highlighted above. Procedure on line 10 takes in C, the set of characters that may make be present in a row. It also takes in the maximum length of a field (such as username), the upper bound on the number of records in the table, number of query retries and the epsilon per query being used to protect the table. The procedure then begins querying for the number of records that begin with the prefix contained in set M in line 18. The 'PerformQuery' procedure is a call to the querying mechanism and returns a noisy count given a query. The query passed is of the form

```
SELECT COUNT(*) FROM table WHERE field LIKE m%
```

This query can be interpreted as retrieving the count of rows that begin with a prefix m and not the character 'm'.

The shape of noise function added is a discretized version of figure 5. The domain of the function is the set of 2^{53} integer multiples of 2^{-53} between 0 and 1. Since the noise is monotone decreasing function, it can be efficiently searched over using binary search(specified in 6). If we fail to find the noisy result along this curve, then we can infer that the count must be non-zero. This is a failure of the mechanism. As Mironov noted in his work, there is always at least a 40% chance of failure with the naive noise generation method so each prefix is queried and then used as input to search until it is either found on the curve or not found after 4 attempts. Binary search finding the original input to the noise even once is an indication of the true count being 0. Thus we can terminate the search and remove this prefix from further consideration. However, if after 4 times (the likelihood of not finding a

Figure 5: Shape of function transforming uniform random variable to laplacian



prefix after 4 queries is 13%) it is not found through the binary search, the prefix is saved to the filtered set N. Finally, each element of the filtered set N is concatenated with every character supported by the table (stylized as the cartesian product on line 26).

The algorithm pseudocode is as follows:

Algorithm 1 Recreate a Database protected by CHORUS

1: **procedure** LAPLACENOISE(x, scale) ▷ Replicates CHORUS' noise generation
2: $u \leftarrow 0.5 - x$
3: $-\text{sign}(u) \cdot \text{scale} \cdot \log(1 - 2|u|)$
4: **end procedure**
5:
6: **procedure** BINARYSEARCH(noisy count, curve)
7: true if noisy count is found on the curve
8: **end procedure**
9:
10: **procedure** RUNNER(C, max length, upper bound, n, ϵ)
11: $M \leftarrow C$ ▷ Initialize set of potential records to the set of characters supported
12: $i \leftarrow 0$
13: scale $\leftarrow \frac{1}{\epsilon}$
14: **while** $i <$ max length **do**

```
15:         for each m ∈ M do
16:             j ← 0
17:             while j < n do
18:                 query result ← PerformQuery(m)
19:                 if BinarySearch(query result, LaplaceNoise(2·scale)) then
20:                     add m to N
21:                     j ← n
22:                 end if
23:                 j ← j + 1
24:             end while
25:         end for
26:         M ← N × C
27:         i ← i + 1
28:     end while
29:     M is set of reconstructed records
30: end procedure
```

4 Results

4.1 Experiment Details

My implementation[1] was in Scala to extend CHORUS smoothly, the table was placed in MySQL and I used a Scala MySQL connector to execute the queries. The database consisted of one table, containing 2000 unique records. Each record was a 4 character prefix of the larger Yelp dataset [13]. The 4 characters were chosen from a set of alphanumeric characters and '-' (63 total possible characters). This was chosen since attacking an equally sized table of 22 character string takes 48 hours to complete.

An experiment variable was the number of query retries. I chose the values 1, 2, 3 and 4; the attack ran 100 times for each of those chosen values. Each query was set to $\epsilon = 10^{-12}$

Finally, I implemented an attack-proof version of the querying mechanism. This is achieved by swapping out the laplace noise generator in line 1 with the snapping mechanism.

4.2 Experiment Results: Attack

In figure 6, each of the boxplots has the number of retries on the x-axis. An interesting note is the presence of the outlier 0% accuracy rate/budget consumed. This is present only in the lower query retry cases. It occurs because a vast majority of the selected records begin with '-'. Thus, the fewer queries issued, the lower the likelihood of finding records. Note that with 4 query retries, there are no instances of 0% accuracy. This 0% accuracy case leads to

[1]Implementation code available https://github.com/AnkitSiva/DPPorosity

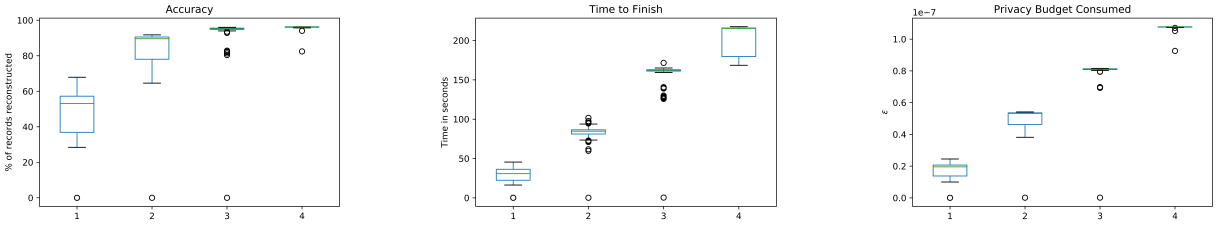


Figure 6: Boxplots of reconstruction rate, time and epsilon consumed

an empty set of potential members, and thus no further queries issued, exiting the program with very little time lost and ϵ consumed on the order of 10^{-12}.

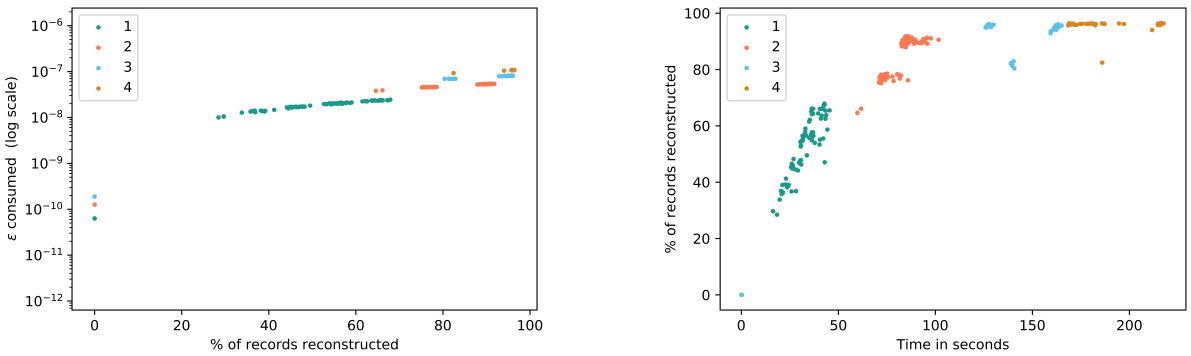


Figure 7: Plots of ϵ consumed against accuracy and accuracy against time

Additionally, looking at the ϵ-accuracy and the time-accuracy trade-offs in figure 7, we see a tapering off in usefulness of every additional query. Quite a few of the attack runs with 3 query retries achieve a similar reconstruction rate as the 4 query retries with closer to 2.5 minute run time than 3.5.

None of these runs use more than $\epsilon = 10^{-7}$. This is particularly frightening as such an attack can be run by an analyst without the budgeting system picking up on a privacy violation. Additional defenses against this attack, without implementing the snapping mechanism would require query scanning (which grows too complex) or limiting the number of queries an analyst can run, alongside their privacy budget.

While these runs have not achieved a 100% record reconstruction ratio, it is easily obtainable by bumping the query retries to a value of 6 or higher. This comes at a time penalty, but very clearly shows that these methods of noise generation are not safe.

4.3 Experiment Results: Defense

Once the snapping mechanism is implemented, the size of potential prefixes grows too large for the current algorithm to discern between neighbors and has not terminated in any runs so

far. Looking at the proof for the snapping mechanism [12], we see that it is in fact impossible to discern between neighboring databases simply off of one query.

This is the safest way to prevent the attack this algorithm poses.

5 Future Work

It is unfortunate that the Uber implementation fell victim to a flaw identified 6 years prior and well known within the privacy community. It is clear that there is a fix, one that is simple enough for an undergraduate to implement! I hope that this thesis brings the attention of future implementers of differentially private algorithms to not ignore important assumptions. Differential privacy unfortunately requires not only an understanding of machine learning and statistics but also an eye for detail similar to the field of security when it comes to implementing these algorithms.

Although the repository has been archived, there are is additional work that can be done towards adaptively reducing the number of retries per query as the length of the prefix grows. Further, new querying strategies like querying just the first character, then the second and then querying the joins can reduce the number of total queries. The application of this algorithm in a multi-table setting also demands investigation.

Acknowledgement

I would like to recognize Dr. Rachel Cummings for her ever-helpful advice, and, Dr. Ling Liu and Dr. Jamie Morgenstern for their guidance towards this project. Others who have provided invaluable theoretical and implementation guidance are Uthaipon (Tao) Tantipongpipat, Justin Kalloor, Samarth Wahal and Shyamal Patel. A heartfelt thank you to everyone that helped make this thesis a reality.

References

[1] Carlini, N., Liu, C., Kos, J., Erlingsson, Ú., and Song, D. The Secret Sharer: Measuring Unintended Neural Network Memorization & Extracting Secrets. *ArXiv e-prints* (Feb. 2018), arXiv:1802.08232.

[2] Cohen, A., and Nissim, K. Linear Program Reconstruction in Practice. *ArXiv e-prints* (Oct. 2018), arXiv:1810.05692.

[3] Cummings, R., and Desai, D. The role of differential privacy in gdpr compliance: Position paper. *Association of Computing Machinery In Proceedings of Workshop on Responsible Recommendation (FATREC'18)* (Oct 2018).

[4] Dinur, I., and Nissim, K. Revealing information while preserving privacy. In *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2003), PODS '03, Association for Computing Machinery, p. 202–210.

[5] Dwork, C., McSherry, F., Nissim, K., and Smith, A. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography* (Berlin, Heidelberg, 2006), TCC'06, Springer-Verlag, p. 265–284.

[6] Dwork, C., and Roth, A. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science 9*, 3–4 (2014), 211–407.

[7] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70.

[8] Johnson, N., Near, J. P., Hellerstein, J. M., and Song, D. Chorus: Differential privacy via query rewriting. *arXiv preprint arXiv:1809.07750* (2018). Accessed on 5 Feb 2020.

[9] Johnson, N., Near, J. P., and Song, D. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment 11*, 5 (2018), 526–539.

[10] McSherry, F. Uber's differential privacy .. probably isn't, Feb 2018. Accessed on 17 Apr 2020.

[11] McSherry, F., and Johnson, N. Differential privacy of example., Feb 2017. Accessed on 17 Apr 2020.

[12] Mironov, I. On Significance of the Least Significant Bits for Differential Privacy. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 650–661.

[13] Yelp, 2014. data retrieved from the Yelp Dataset Challenge.