Simulatable Auditing: Databases Final Project

Serena Chen

Spring 2019

1 Statistical Databases

Statistical databases are those that allow aggregate queries but not access to individual records. These are used in contexts where the database owner wants to allow outside developers and statisticians to study broad trends in their database, but doesn't want outsiders to access sensitive or identifying information. Examples of public use statistical databases allow users to access census data[1] or about cancer incidence[3].

In addition to only allowing aggregate queries, these statistical databases often have other security measures to prevent identification. These include, things like adding noise to the output or sampling randomly from the set of records queried[5]. For large, public database like the census, there are enough data points to make accessing identifying information difficult. In addition, the Census Bureau has a specific query language that only allows querying predefined sets of records.

2 Online Auditing

Another way to prevent identifying information from being leaked is a process called *online auditing*. For a statistical database operating under the constraint that each query made must either be answered truthfully or denied, online auditing determines how to answer a query such that no sensitive information is leaked by reviewing the history of queries and answers [1]. This is a specific instance of the inference control problem, which broadly tries to figure out how to prevent attackers from being able to access information they are not supposed to have.

It's unclear how much online auditing is actually used in real applications. Many public databases are (very reasonably) not open with the security measures they take. I also suspect that this is more often used for smaller, enterprise databases, since it seems many public use databases are designed such that no possible query could leak sensitive information. I also suspect that some databases are less strict on the constraint to only truthfully respond or deny. There has been a lot of fairly recent work on differential privacy, which introduces noise to the response to hide sensitive data.

For this project, I'm using a few constraints that are maybe not super representative of the real world.

The first is that we can query the database by specifying a set of primary keys. In many (but probably not all) of these statistical databases, you would not normally be able to specify which exact records you aggregate. Instead, you would aggregate based on broader attributes, such as gender, race, job title, geographical location, etc. Querying based on certain primary keys makes it significantly easier for me to demonstrate security breaches. I think that the scenarios here are still applicable in databases that aggregate

[1] In case you're wondering, offline auditing also exists, and is the process of looking through a history of queries and answers and determining whether a breach has occurred.

based on broader select clauses, where the user does not explicitly know which specific records are being aggregated. However, there is still a small jump to get from one to the other.

The second constraint is that I'm going to focus on the implementations of auditing that only deal with max queries. Auditing gets a lot more complicated with more types of aggregate queries. It was shown that the online auditing problem for sum and max queries is NP-hard[2]. Comparing max auditing techniques seemed in scope and something that I could reasonably understand. Although, honestly, if I knew then what I knew now, I would probably look more broadly at the space of inference control. This was still a lot of fun and a good exercise, though.

3 Early Work on Online Auditing

Work online auditing goes back to the 70's and 80's. In the 80's, a basic algorithm for auditing a series of max queries was proposed[2]. It works by taking the history of queries and answers and building up something they call the "knowledge space." The knowledge space includes every query and answer up to that point as well as every query and answer that can be inferred. These inferences are programmatically generated using a set of transformations derived from set properties[2]:

1. Given two queries $max(s_1) = m_1$ and $max(s_2) = m_2$,
   - if $m_1 = m_2$, add $max(s_1 \cap s_2) = m_1$ and $max(s_1 \oplus s_2) < m_1$ to the knowledge space
   - if $m_1 > m_2$, add $max(s_1 - s_2) = m_1$ to the knowledge space
2. Given two queries $max(s_1) = m_1$ and $max(s_2) < m_2$,
   - if $m_1 \geq m_2$, add $max(s_1 - s_2) = m_1$ to the knowledge space
   - if $m_1 < m_2$, add $max(s_2 - s_1) < m_2$ to the knowledge space
3. Given two queries $max(s_1) < m_1$ and $max(s_2) < m_2$,
   - if $m_1 = m_2$, add $max(s_1 \cup s_2) < m_1$ to the knowledge space
   - if $m_1 > m_2$, add $max(s_1 - s_2) < m_1$ to the knowledge space

The full knowledge space is generated by iteratively applying all of these transformations until there are no more. When a new query is made, that query and it's answer is added to the knowledge space, and the transformations are iteratively applied again.

If adding the new query to the knowledge space and applying the transformations causes a query in the knowledge space of the form $max(\{x\}) = m$ for some single element x, we know that an attacker will be able to infer the specific value for row x. As a result, we should deny the request.

[2]A bit like De Morgan's laws but for query sets

3.1 Demonstration of the algorithm

Lets say we have the following database:

ID	Name	Salary
0	Bob	60000
1	Erin	80000
2	Marty	70000
3	Abby	70000

and we have the end user makes the following two queries:

- $max(ID = \{0, 1, 2, 3\})$ (answer is 80000)
- $max(ID = \{0, 2, 3\})$ (answer is 70000)

Here we have a fairly straightforward information leak. From the first query, we know that there must be one element out of the four that has a salary of \$80000. However, upon querying the max of three of the four elements, we know that those three people make \$70000 or less. Thus, we can deduce that the one element we didn't put in the second query must be the one with a salary of \$80000.

What happens when you use the above auditing technique?

Well, for the first query, the knowledge space only contains that query ($max(\{0, 1, 2, 3\}) = 80000$), so no information has been leaked, and 80000 is returned.

On the second query, the knowledge space looks like this:

- $max(\{0, 1, 2, 3\}) = 80000$
- $max(\{0, 2, 3\}) = 70000$

Applying the first transformation rule, we add a value to our knowledge space.

- $max(\{0, 1, 2, 3\}) = 80000$
- $max(\{0, 2, 3\}) = 70000$
- $max(\{1\}) = 80000$

Thus, the knowledge space shows that the value for ID=1 can be inferred from the second query, so the auditor denies the query.

3.2 Security flaw in the algorithm

In the previous example, we queried:

- $max(ID = \{0, 1, 2, 3\})$ (answer is 80000)
- $max(ID = \{0, 2, 3\})$ (answer is 70000, our auditor would deny)

But let's say we used these queries instead:

- $max(ID = \{0, 1, 2, 3\})$ (answer is 80000)
- $max(ID = \{0, 1, 2\})$ (answer is 80000, auditor behavior is ??)

Disregarding the responses to all the queries, these two sets of queries seem very similar on the surface. They both query the max of four elements, and then query the max of three out of the four elements.

How would the auditor respond to the second set? Well, the first query would always respond successfully. Upon receiving the second query, the knowledge space would look like:

- $max(\{0, 1, 2, 3\}) = 80000$
- $max(\{0, 1, 2\}) = 80000$

If we again apply the first transformation, the knowledge space would be expanded to this:

- $max(\{0, 1, 2, 3\}) = 80000$
- $max(\{0, 1, 2\}) = 80000$
- $max(\{0, 1, 2\}) = 80000$ (in this case, a duplicate)
- $max(\{3\}) < 80000$

If we were to continue applying transformations, we would find that there is nothing new to add to our knowledge space. Since there is nothing in the knowledge space of the form $max(\{x\}) = m$, the auditor would determine that there is no security risk and would return the proper value instead of denying.

This seems reasonable. Saying that $max(\{0, 1, 2\}) = 80000$ doesn't reveal anything identifying about any of the elements of the database. However, this is very different behavior to the first set of queries. Even though the two sets of queries seem very similar, one denies the second query and the other returns 80000. If an attacker knew how the auditor worked, or even had a little bit of intuition based on previous experiences, they could reason that the first set of queries didn't work because the auditor saw a way to infer a specific data point. However, the only way that information could be leaked from the first set of queries is if ID=1, the only ID not included in the second query, is the max and all the other IDs have values smaller than that. Thus, denying the request will still tell the attacker that the person with ID=1 has a salary of $80000.

4 Simulatable Auditing

Simulatable auditing came out of that exact problem with earlier auditing algorithms[4]. The earlier auditing algorithms leak information because they use information that the attacker doesn't know (the answer to the newest query) in order to decide whether to answer or deny. Therefore, an attacker can infer what the actual response to the query was that led the auditor to believe there may be a compromise.

Simulatable auditors are named as such because they are auditors that are fully simulatable by the attacker. They do not leak any new information because it does not use any information that the attacker would not know beforehand. Thus, an attacker would be able to predict the result of the auditor.

One way of implementing a simulatable auditor is to take the new query, generate every possible answer the database could give to that query based on the queries that have come before it, and if any of those answers would cause an individual's information to be uniquely identified, then deny the query.

In the context of max queries, the algorithm would look roughly like this:

- Receive the new query $max(q)$
- Find all previous queries that overlap with q, and put all the answers into a set M.
- For each two adjacent elements in sorted(M), take the midpoint between those two numbers and add the result to M.
- Take the smallest element s in M, and put $s-1$ in M. Then take the largest element b in M, and put $b+1$ in M.
- For each $m \in M$, if m is *consistent* with previous answers AND setting m as the response to $max(q)$ uniquely identifies any element in the database, deny.

That last step is pretty vague, so let's break down what it means for an answer to be consistent or for an element to be uniquely identifiable.

Consistency means that every previous query made is still valid. We can model each element as having an upper bound based on the lowest max that the database has returned for that element thus far. For example, if the previous queries were $max(\{0,1\}) = 80000$ and $max(\{1,2\}) = 70000$, then the mapping would be {0:80000, 1:70000, 2:70000}.

If the database previously said that $max(\{a_1, ..., a_n\}) = x$, then at least one of $a_1, ..., a_n$ must have value x, and therefore must have an upper bound of x. If a new query says that $max(\{a_1, ..., a_n\}) = x - 1$, then an inconsistency occurred. The first query states that some element in $a_1, ..., a_n$ has value x, but the second query states all of $a_1, ..., a_n$ is less than or equal to $x - 1$, a contradiction.

Thus, an easy way to check for inconsistency is to iterate through each previous query and check that the there is still at least one element in the query's query set with an upper bound that matches the answer to the query.

For a record to be uniquely identifiable, there must be only one element for a given query whose upper bound matches the query's answer.

4.1 Revisiting 3.2

You will recall that we previously had two sets of queries:

- $max(ID = \{0,1,2,3\})$ (answer is 80000)
- $max(ID = \{0,2,3\})$ (answer is 70000)

and

- $max(ID = \{0,1,2,3\})$ (answer is 80000)
- $max(ID = \{0,1,2\})$ (answer is 80000)

You will also recall that the old auditor would deny the first set while answering the second set. The denial of the first one indicates that the auditor knew that responding truthfully would lead to sensitive data being leaked. However, since the only scenario in which information would be leaked was if ID=1 was the max with 80000, the attacker is able to gain this information even despite the denial.

Let's walk through what would happen with the new simulatable auditor. In both cases, the first query is answered, as there is no previous information to consider.

For the second query, the simulatable auditor would actually work the same way in both cases. The auditor would look through all previously answered queries that overlap with the new one (in our case, it is the only other query that was made). The auditor would then generate a set with all the answers of the identified queries ({80000}). Then it would find all the midpoints between two consecutive answers. This step doesn't apply in our case. Then you add in your lower and upper bounds, which are the lowest element of the set minus one and the highest element of the set plus one. In our case, we add 79999 and 80001. Then, for each of those values, we set the answer to the new query equal to that value and check if any records have been compromised.

In our case, when we use 79999 as the answer to the second query, both cases will detect a breach. In the first case, the upper bounds are {0:79999, 1:80000, 2:79999, 3:79999}, and in the second case, the upper bounds are {0:79999, 1:79999, 2:79999, 3:80000}. In each of these, one element is uniquely identified to have a value of 80000, so in both cases, the second query is denied.

In this way, denials are guaranteed to not leak any information. You might notice that now we are denying in the second case, which on its own, does not leak any individual records. Thus there is a tradeoff between security and being able to provide as much information as possible. This tradeoff would probably be lessened if there were more options to deal with queries than to andwer truthfully or not at all.

References

[1] US Census Bureau. *Developers*. URL: `https://www.census.gov/developers/` (visited on 05/10/2019).

[2] Francis Chin. "Security Problems on Inference Control for SUM, MAX, and MIN Queries". In: *J. ACM* 33.3 (May 1986), pp. 451–464. ISSN: 0004-5411. DOI: `10.1145/5925.5928`. URL: `http://doi.acm.org/10.1145/5925.5928`.

[3] Centers for Disease Control and Prevention. *NPCR and SEER Incidence – U.S. Cancer Statistics Public Use Databases*. URL: `https://www.cdc.gov/cancer/uscs/public-use/index.htm` (visited on 05/10/2019).

[4] Krishnaram Kenthapadi, Nina Mishra, and Kobbi Nissim. "Simulatable Auditing". In: *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '05. Baltimore, Maryland: ACM, 2005, pp. 118–127. ISBN: 1-59593-062-0. DOI: `10.1145/1065167.1065183`. URL: `http://doi.acm.org/10.1145/1065167.1065183`.

[5] Arie Shoshani. "Statistical Databases: Characteristics, Problems, and Some Solutions". In: *Proceedings of the 8th International Conference on Very Large Data Bases*. VLDB '82. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1982, pp. 208–222. ISBN: 0-934613-14-1. URL: `http://dl.acm.org/citation.cfm?id=645910.673595`.