Secure Cloud Computing:
Encrypted Databases and Searchable Symmetric Encryption

Riccardo Gennaro
Student ID: 3534219
Group 5

Péter Svelecz
Student ID: 3542629
Group 5

1. Proving Information Leakage of an SSE Scheme

1.1 Approach for Proving Information Leakage in SSE Schemes

As described in the assignment text, to prove an upper bound for the information leaked, or to quantify the information leakage for an SSE scheme, we base our approach on simulation-based experiments as per Curtmola et al. In particular, we base our approach on simulation to prove non-adaptive semantic security.

We perform the following:

- **Define the Leakage Function L**: the leakage function expresses which information is exposed to an adversary when executing a protocol based on the given SSE schema. In particular, this function specifies the nature of the leaked information, e.g. number of documents, access frequencies, etc. This leakage is then given as input to the simulator.
- **Construct a simulator S for the protocol with the stated leakage L**: the simulator is an algorithm that simulates the adversary view of the real protocol. This algorithm takes as input the leakage of the SSE schema and simulates an output that needs to appear indistinguishable from the output of the execution of the real protocol.
- **Prove indistinguishability of the execution from the adversary perspective**: the adversary (or distinguisher) sends the challenger a set of queries. The challenger execute either the real or the simulated protocol. Indistinguishability is proven if the adversary is not able to distinguish between the real and the simulated output with a negligible advantage ϵ .

1.2 Leakage statement

The provided SSE scheme presents the following elements in the leakage:

- **Number of documents** $|\mathcal{D}|$: since by construction documents D_i are mapped one-to-one to encrypted keywords sets C_i, the number of C_i sets is equal to the number of documents. Since the index is $\mathcal{I} = C_1, ..., C_d$, where d is the number of documents, the number of documents is leaked.
- **Number of keywords per document** $n_i = |C_i|$: since the index $\mathcal{I}$ contains the set of encrypted keywords C_i, the number of keywords associated to a document C_i is leaked.
- **Keyword equality - number of distinct keywords**: since we the keywords are encrypted using a deterministic pseudorandom function $F : 0,1^{\lambda} \times 0,1^{*} \rightarrow 0,1^{n}$, for equal keywords $p_i = p_j$ the pseudorandom function F will return equal results $\gamma_i = F(k, p_i) = F(k, p_j) = \gamma_j$. This leaks the number of distinct keywords.

- **Search pattern** $SP(\mathbf{q})$: the adversary can determine what queries are for the same keyword since the search tokens τ_q are deterministically computed using pseudorandom permutation F.
- **Access pattern** $AP(\mathbf{q})$: the adversary can determine what document identifiers are returned for a given query q_i as query result $D(q_i)$.

1.3 Proof sketch

Given the access pattern $AP(\mathbf{q})$, for all sets $D(q_i) \in AP(\mathbf{q})$ with i s.t. $1 \leq i \leq q$, for every document identifier $j \in D(q_i)$, insert in the set of encrypted keywords C_j of document D_j the simulated encrypted keyword γ'_j sampled uniformely and randomly as $\gamma'_j \xleftarrow{\$} \{0,1\}^n$, instead of being computed with pseudorandom permutation F. As a result, we obtain simulated index $\mathcal{I}' = C'_1, ..., C'_d$,

For every $q_j \in SP(\mathbf{q})$, set search token $\tau'_j = \gamma'_j$ instead of computing it with pseudorandom permutation F.

We compare the simulated parameters with the real ones:

- $\mathcal{I}$ **and** $\mathcal{I}'$: $\mathcal{I}$ is the collection $C'_1, ..., C'_d$ containing the sets of the keywords of every document encrypted with deterministic pseudorandom permutation F. If collection $AP(\mathbf{q})$ is non-empty, $\mathcal{I}'$ will contain the collection $C'_1, ..., C'_d$ of the sets of the simulated keywords of the documents indexed in $AP(\mathbf{q})$. The simulated encrypted keywords are sampled uniformly and randomly and with length n, matching the length of the real keywords. Since the key used with F is not known by the distinguisher, it is not able to distinguish C_i from C'_i.
- τ_j **and** τ'_j: τ_j is computed via the deterministic F function and its value will match one of the real encrypted keywords. The value of τ'_j is assigned in such a way that for query q_j in $SP(\mathbf{q})$ it will be equal to simulated encrypted keyword γ'_j.

Since the structure is preserved, the only way for a distinguisher to distinguish between real and simulated parameters is to be able to differentiate between the output of the random and uniform sampling and the output of the PRP function that is considered secure.

2. Attacking Property-preserving Encryption

2.1 Number of unique first names in the database

In the database there are a 100 unique first names stored. This number was determined by taking into consideration that the names in the database were encrypted using deterministic encryption. As a consequence, the number of unique encrypted first names is equivalent to the number of first names in the database.

Listing 1: Java Code to Count Unique First Names

```
1   public int getFirstNameCount() {
2       String query = "SELECT COUNT(DISTINCT enc_firstname) FROM
            enc_students";
3
4       try (Connection conn = DriverManager.getConnection(url);
5            PreparedStatement pstmt = conn.prepareStatement(query)
                 ;
6            ResultSet rs = pstmt.executeQuery()) {
7
8           if (rs.next()) {
9               return rs.getInt(1);
10          }
11
12      } catch (SQLException e) {
13          System.out.println("Error: " + e.getMessage());
14      }
15      return 0;
16  }
```

2.2 Distribution of last names

To plot the distribution of the encrypted last names we determined the number of occurrences of each last name present in the database.

Following, the code to compute the lastname frequency.

Listing 2: Distribution of last names

```
17
18  public HashMap<String, Integer> getLastnameFrequenciesOrdered()
        {
19      String query = "SELECT enc_lastname FROM enc_students";
20
21      try (Connection conn = DriverManager.getConnection(url);
22           PreparedStatement pstmt = conn.prepareStatement(query)
                 ;
23           ResultSet rs = pstmt.executeQuery()) {
24
25          HashMap<String, Integer> lastnameFrequencies = new
                HashMap<>();
26          while (rs.next()) {
27              String lastname = rs.getString("enc_lastname");
28              if (lastnameFrequencies.containsKey(lastname)) {
29                  lastnameFrequencies.put(lastname,
                        lastnameFrequencies.get(lastname) + 1);
30              } else {
31                  lastnameFrequencies.put(lastname, 1);
```

```
32                  }
33              }
34
35              return orderHashMapByValue(lastnameFrequencies);
36
37          } catch (SQLException e) {
38              System.out.println("Error: " + e.getMessage());
39          }
40          return null;
41      }
```

Following, the graph with lastnames in chipertext.

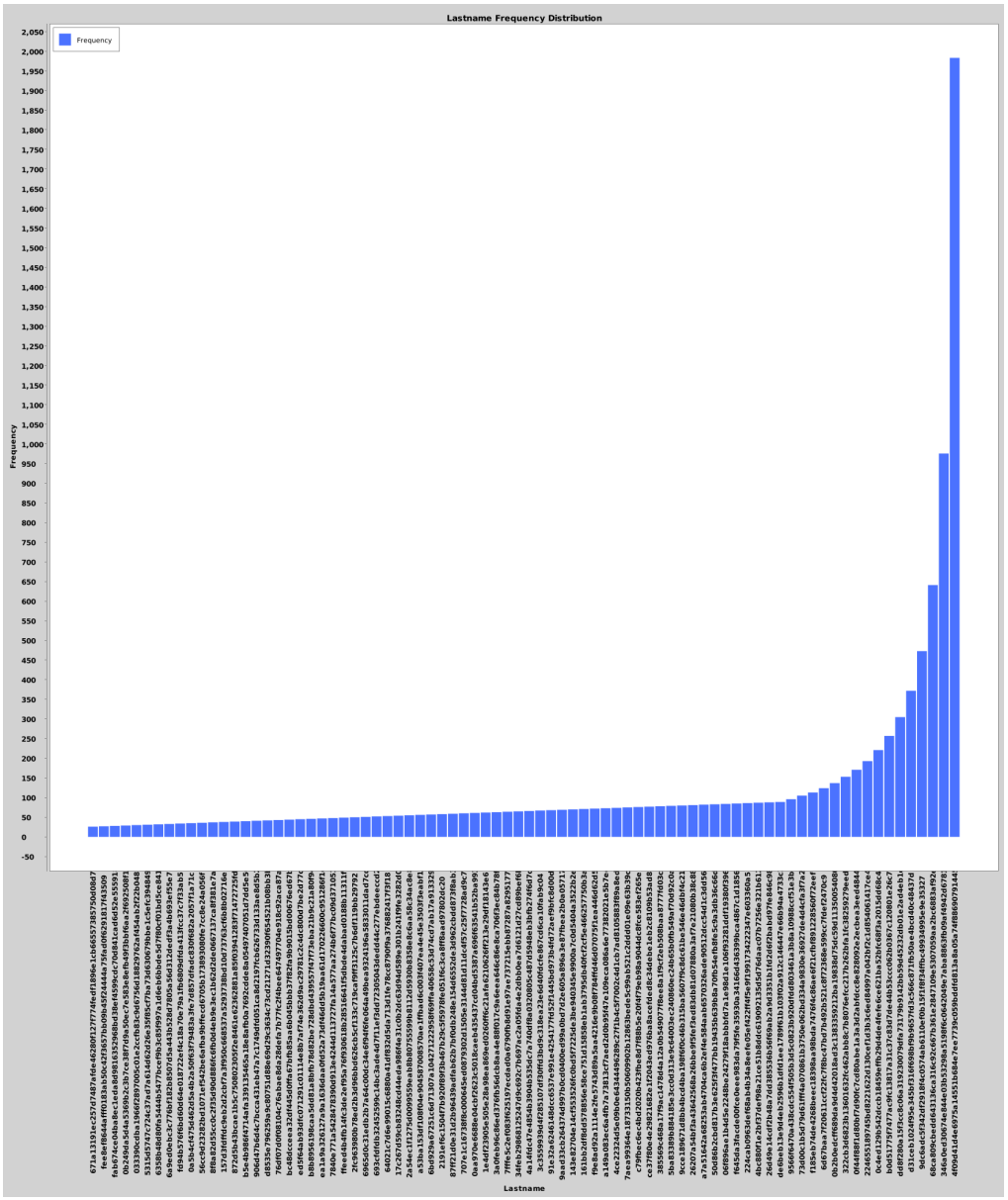

Figure 1: Distribution of encrypted last names.

The graph shows the distribution of the encrypted last names names. The distribution is clearly non-uniform, and an attacker can use background information (like the files given for this assignment) to infer by crafting a lookup table for mapping encrypted values to plaintext values.

Following, the same graph with names in plaintext.

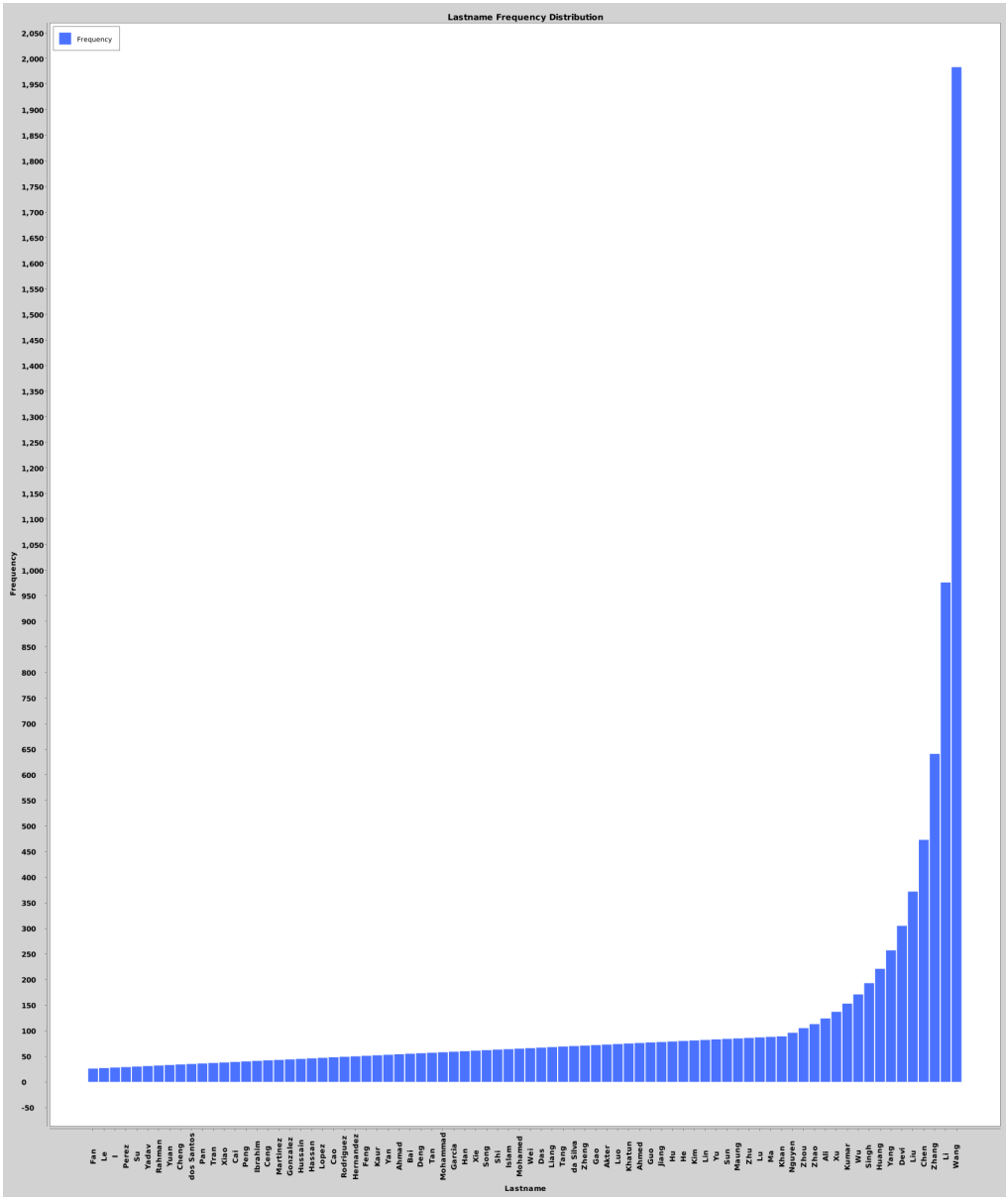

Figure 2: Distribution of encrypted last names.

As it can be seen, lastname Wang (which is the most popular last name) is mapped to the chipertext
`4f09d41d4e6975a14551b684e7ee7739c059bddfd813a8a05a74f88690791445`.

2.3 Full names of all students who scored 99 points

There are 100 unique values stored in the database for grades. We also know that the grades can range from 1 to 100. This means that the data set for the grades is a dense dataset since the complete plaintext space is encrypted. Since the grades are encrypted in an order-preserving manner, the lower numeric value of an encrypted grade corresponds to a lower plaintext value. Based on this information, we sorted the ciphertexts decreasingly, this is how we were able to recover the ciphertext value corresponding to 99.

Following that, we queried all the records where the value of the grade was the previously determined one and we got the students who scored 99 points. The encrypted names could be mapped to the plaintext ones based on the given txt files (again, by leveraging the use of deterministic encryption).

We realized that when the encrypted grades and the number of their occurrences are queried, not all the values of the enc_grade column could be parsed as an integer, and also that without casting the column as a numeric type SQLITE applied lexicographic ordering since it handled the data as strings instead of numbers and using numeric ordering. To overcome this, we cast the enc_grade column to a numeric value, so the ordering was done correctly.

In total there were 4 students who scored 99 points, their names are the following: Mario Li, Qing Khan, Ana Jiang, and Maria Li.

Listing 3: Querying for the students who got 99

```
42 public List<Student> getSecondToTopStudents() {
43         String secondValue = getSecondUniqueHighestValue();
44         String query = "SELECT enc_firstname, enc_lastname FROM
               enc_students WHERE cast(enc_grade as real) = ("
45                 + "    SELECT DISTINCT cast(enc_grade as real) as
                       enc_grade_real"
46                 + "    FROM enc_students"
47                 + "    ORDER BY enc_grade_real DESC LIMIT 1 OFFSET
                       1);";
48         List<Student> resultList = new ArrayList<>();
49
50         try (Connection conn = DriverManager.getConnection(url);
51              PreparedStatement pstmt = conn.prepareStatement(query
                    ) {
52             ResultSet rs = pstmt.executeQuery();
53
54             ResultSetMetaData metaData = rs.getMetaData();
55             int columnCount = metaData.getColumnCount();
56
57             while (rs.next()) {
58                 Student student = null;
59                 for (int i = 1; i <= columnCount; i++) {
60                     student = new Student(
61                             rs.getString("enc_firstname"),
62                             secondValue,
63                             rs.getString("enc_lastname")
64                     );
65                 }
66                 resultList.add(student);
67             }
68
69         } catch (SQLException e) {
```

```
70              System.out.println("Error: " + e.getMessage());
71          }
72
73          return resultList;
74  }
```

2.4 Implementing (α, t)-secure index as mitigation

The main idea behind the (α, t)-secure index is to protect against inference attacks by adding noise to the data in a structured way via clustering the result sets into different partitions. The α parameter specifies the minimum number of keywords that must share a similar access pattern. A higher α means more keywords will be grouped together, increasing privacy but possibly introducing more noise. The t parameter indicates the maximum allowable Hamming distance (i.e., the number of differing bits) between the access patterns of keywords within the same group. A smaller t ensures more similarity between access patterns, improving privacy but potentially leading to more false positives during search results.

We implemented a $(4, 0)$-secure index by a lookup function, which when searched for by their last names, returns students in clusters of 4. This leads to confusing results when an adversary tries to map the distribution of names similarly to how it was described previously, but it also introduces 3 false-positive results per query, with which the client executing the query has to deal.

Following, the graph of lastnames' frequencies after implementing a $(4, 0)$-secure index search. As it can be seen, the previous easily mappable distribution is gone, instead clusters of 4 appeared.

When looking at the plaintext equivalents of the last names from the previous graph, it can also be observed, that not only the distribution changed, but so did the few most frequent last names too.

Following, the code

Listing 4: secure index as mitigation

```
75
76      public List<Student> getSecStudentsFromSurname(String surname,
            int alpha) {
77          List<Student> students = new java.util.ArrayList<>();
78          int i = surnameList.indexOf(surname);
79
80          //compute cluster boundaries
81          int clusterUpperBound = i;
82          for (int j = i; (j % alpha != 0 || j==i) && (j<rows); j++)
                {
83              clusterUpperBound = j;
84          }
85          int clusterLowerBound = i;
86          for (int j = i; (j % alpha != 0) && j>0 ; j--) {
87              clusterLowerBound = j-1;
88          }
89
90          //return students in alpha-size cluster
91          if(i != -1) {
92              for (int j = i; j <= clusterUpperBound && (j<rows); j
                    ++) {
93                  for (int k = 0; k < cols; k++) {
94                      if (matrix[j][k]) {
95                          students.add(studentList.get(k));
```

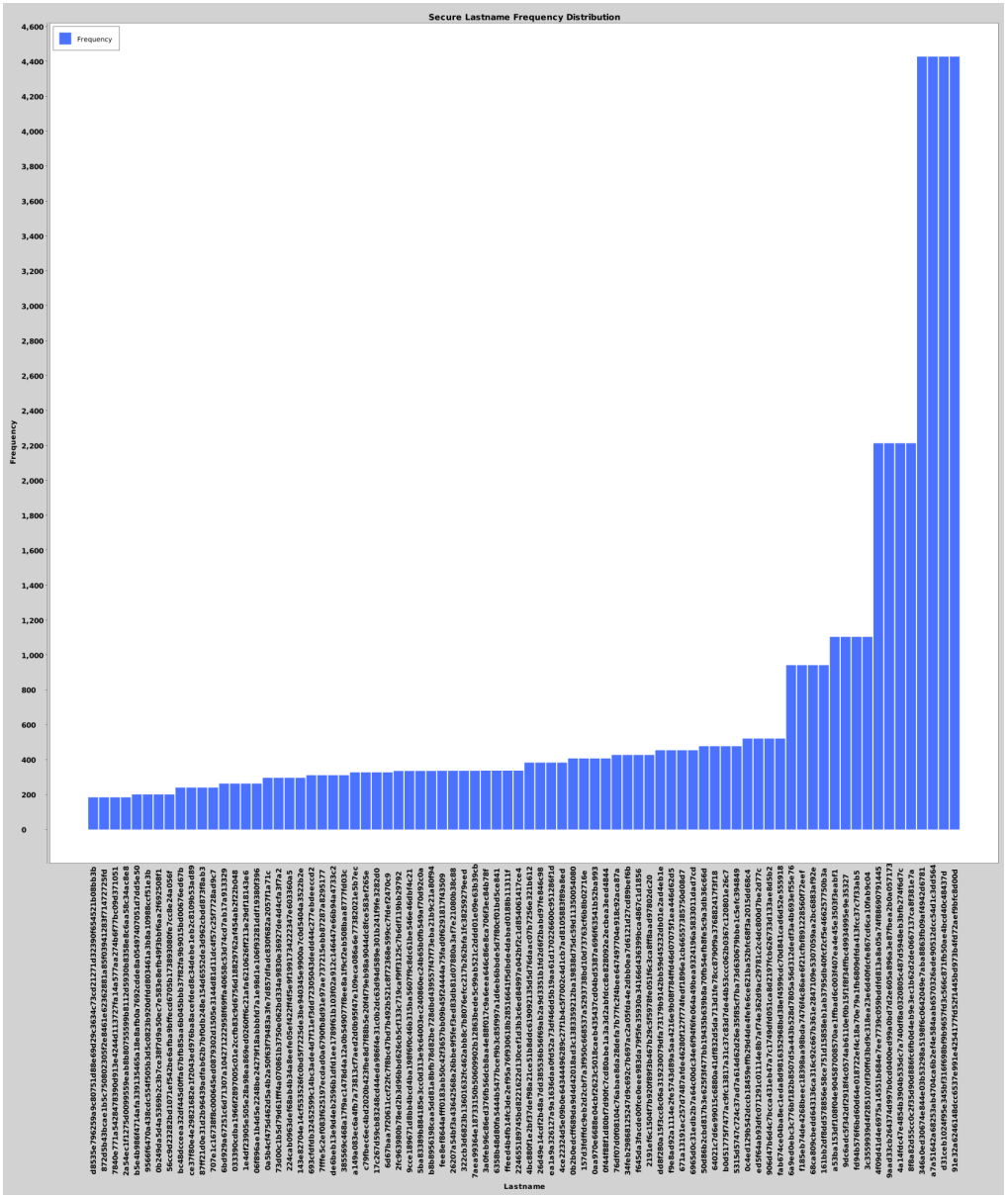

Figure 3: Distribution of encrypted last names.

```
96                          }
97                      }
98                  }
99                  i--;
100                 for (int j = i; j >= clusterLowerBound; j--) {
101                     for (int k = 0; k < cols; k++) {
102                         if (matrix[j][k]) {
103                             students.add(studentList.get(k));
104                         }
105                     }
106                 }
107         }
```

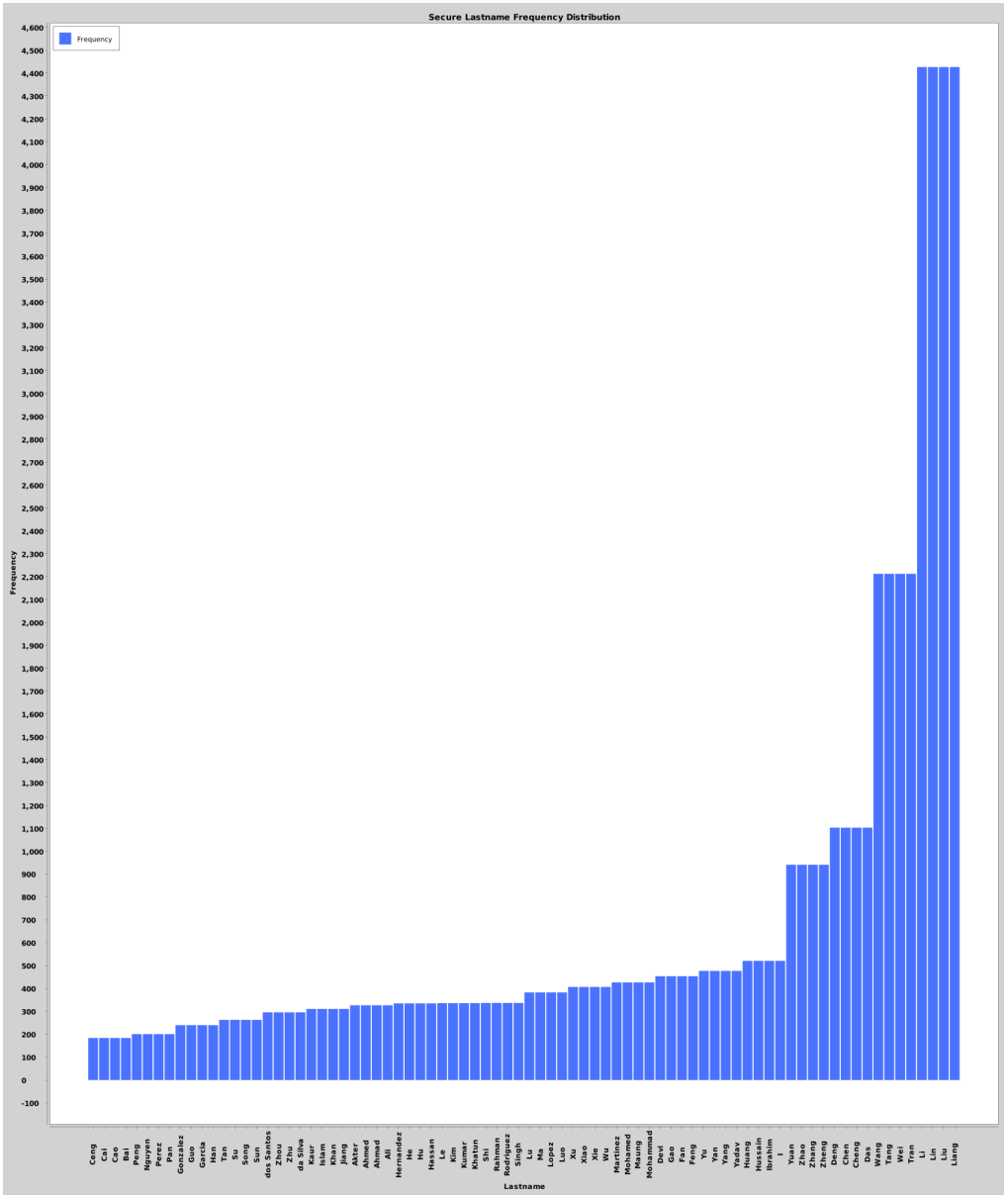

Figure 4: Distribution of encrypted last names.

```
108
109             return students;
110         }
```