Secure Cloud Computing:
Encrypted Databases and Searchable Symmetric Encryption

Riccardo Gennaro
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

Péter Sveleczz
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

1. Proving Information Leakage of an SSE Scheme

1.1 Approach for Proving Information Leakage in SSE Schemes

As described in the assignment text, in order to to prove an upper bound for the information leaked, or to quantifying the information leakeage for an SSE scheme, we base our approach on simulation-based experiments as per Curtmola et al. [**article:curtmola**]. In particular, we base our approach on the simulation for proving non-adaptive semantic security.

We perform the following:

- **Define the Leakage Function L**: the leakage function expresses which information are exposed to an adversary during the execution of a protocol based on the given SSE schema. In particular, this function specify the nature of the leaked information, e.g. number of document, 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 leakeage of the SSE schema and simulate 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) send the challenger a set of query. 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 present 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 is able to 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 is able to 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 encripted keywords are sampled uniformely and randomly and with lenght 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 distinguish 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 are stored 100 unique first names. 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 encripted 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

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.

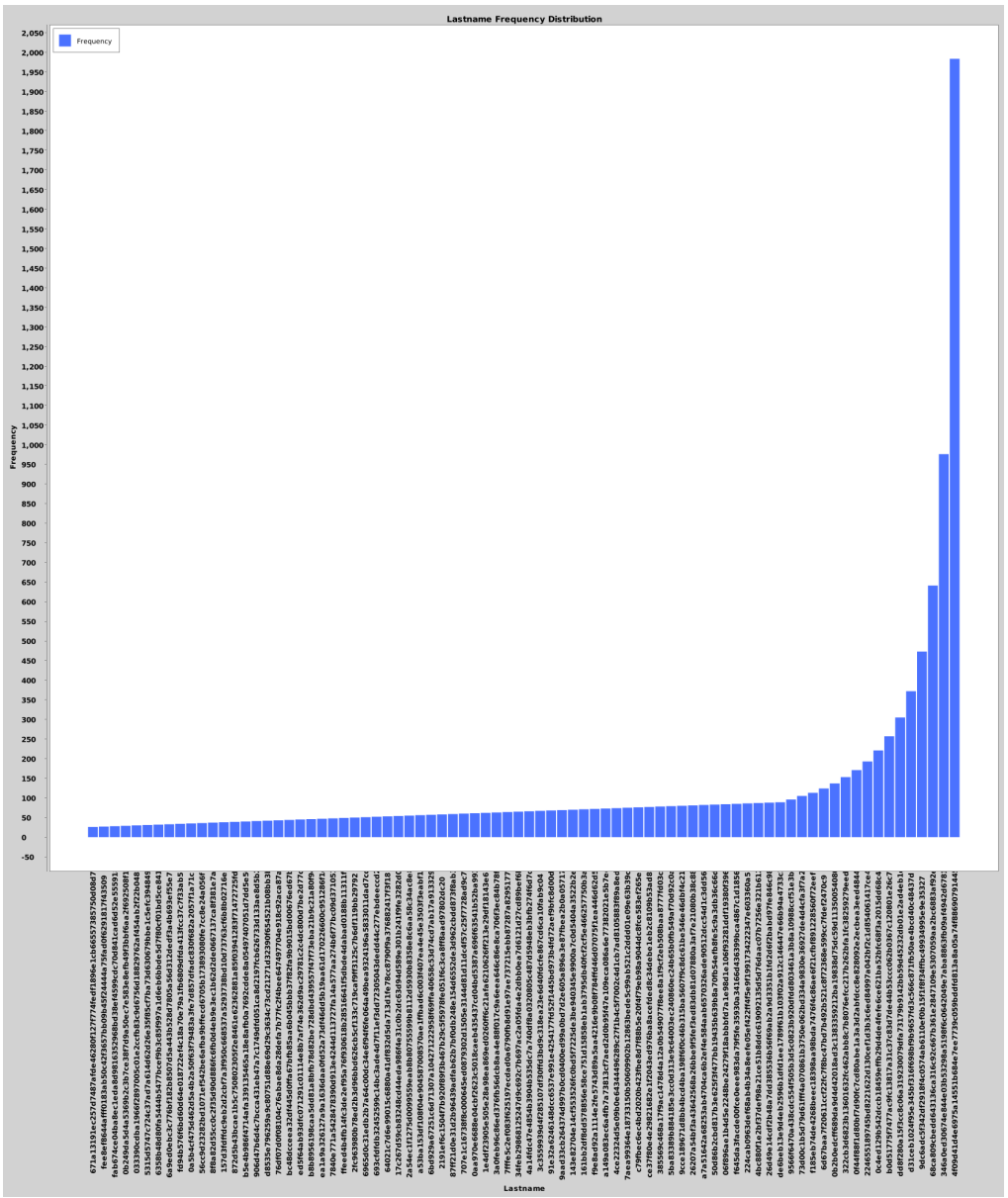

Figure 1: Distribution of encrypted last names.

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

values to plaintext values.

Following, the same graph with names in plaintext.

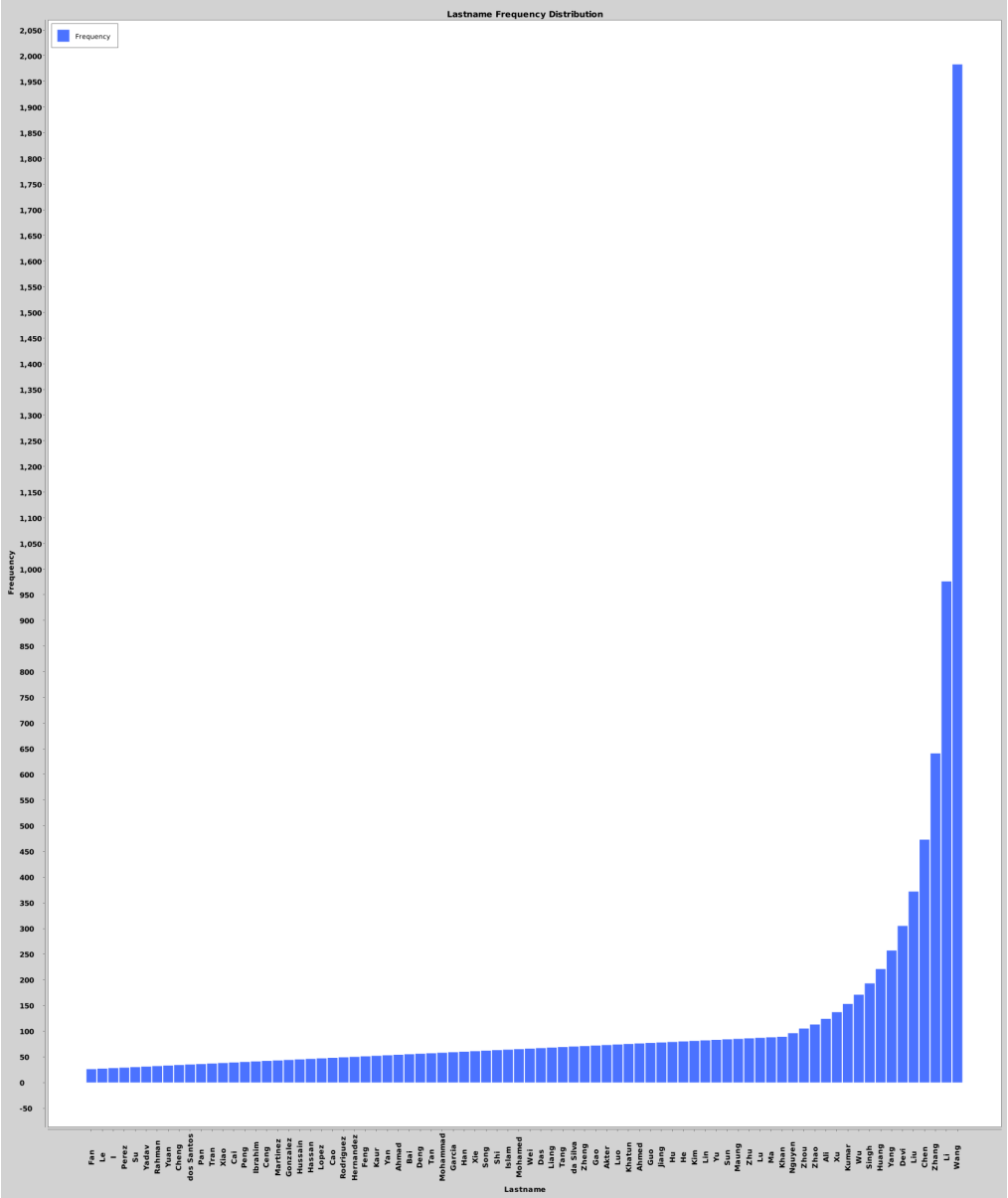

Figure 2: Distribution of encrypted last names.

As can be seen, lastname Wang is mapped to chipertext `4f09d41d4e6975a14551b684e7ee7739c059bddfd813a8a05a74f88690791445`.

2.3 Full names of all students who scored 99 points

Listing 3: Querying for the students who got 99

```
42 public String getSecondUniqueHighestValue() {
43         String query = "SELECT enc_grade FROM ("
44                 + "    SELECT DISTINCT enc_grade"
```

```
45                  + "    FROM enc_students"
46                  + "    ORDER BY enc_grade DESC"
47                  + ") as eseg LIMIT 1 OFFSET 1;";
48
49          try (Connection conn = DriverManager.getConnection(url);
50               PreparedStatement pstmt = conn.prepareStatement(query)
                     ) {
51
52              // Execute the query
53              ResultSet rs = pstmt.executeQuery();
54
55              // Process the result
56              if (rs.next()) {
57                  return rs.getString("enc_grade");
58              }
59          } catch (SQLException e) {
60              System.out.println(e.getMessage());
61          }
62          return null;
63      }
64
65      public List<Student> getSecondToTopStudents() {
66          String secondValue = getSecondUniqueHighestValue();
67          String query = "SELECT enc_firstname, enc_lastname FROM
                enc_students WHERE enc_grade = ?";
68          List<Student> resultList = new ArrayList<>();
69
70          try (Connection conn = DriverManager.getConnection(url);
71               PreparedStatement pstmt = conn.prepareStatement(query)
                     ) {
72
73              pstmt.setString(1, secondValue);
74              ResultSet rs = pstmt.executeQuery();
75
76              ResultSetMetaData metaData = rs.getMetaData();
77              int columnCount = metaData.getColumnCount();
78
79              while (rs.next()) {
80                  Student student = null;
81                  for (int i = 1; i <= columnCount; i++) {
82                      student = new Student(
83                              rs.getString("enc_firstname"),
84                              secondValue,
85                              rs.getString("enc_lastname")
86                      );
87                  }
88                  resultList.add(student);
89              }
90
91          } catch (SQLException e) {
92              System.out.println("Error: " + e.getMessage());
93          }
94
95          return resultList;
96      }
```

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

Following, the graph of lastnames' frequencies after implementing a $(4, 0)$-secure index search.

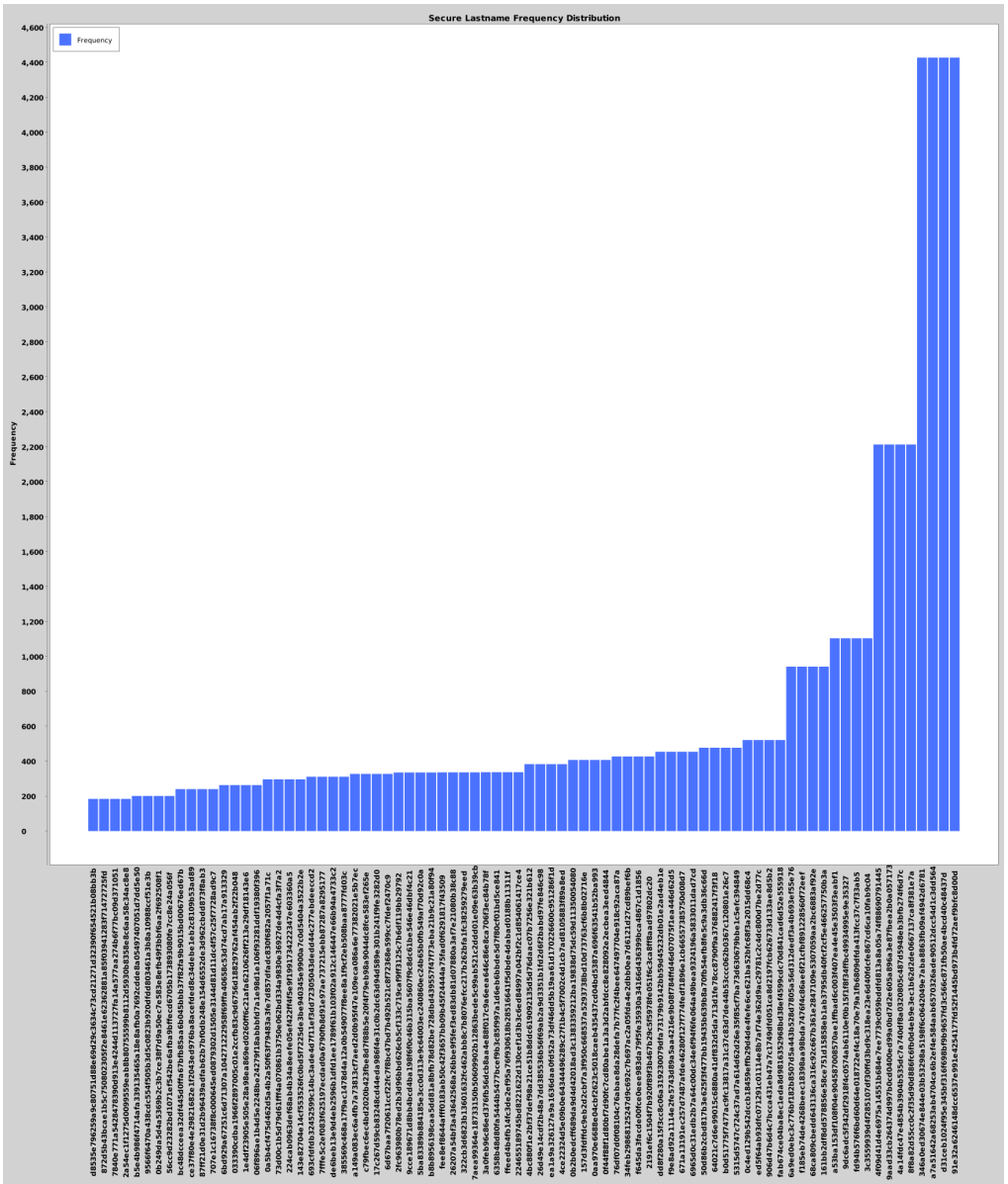

Figure 3: Distribution of encrypted last names.

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

Following, the same graph with names in plaintext.

Following, the code

Listing 4: secure index as mitigation

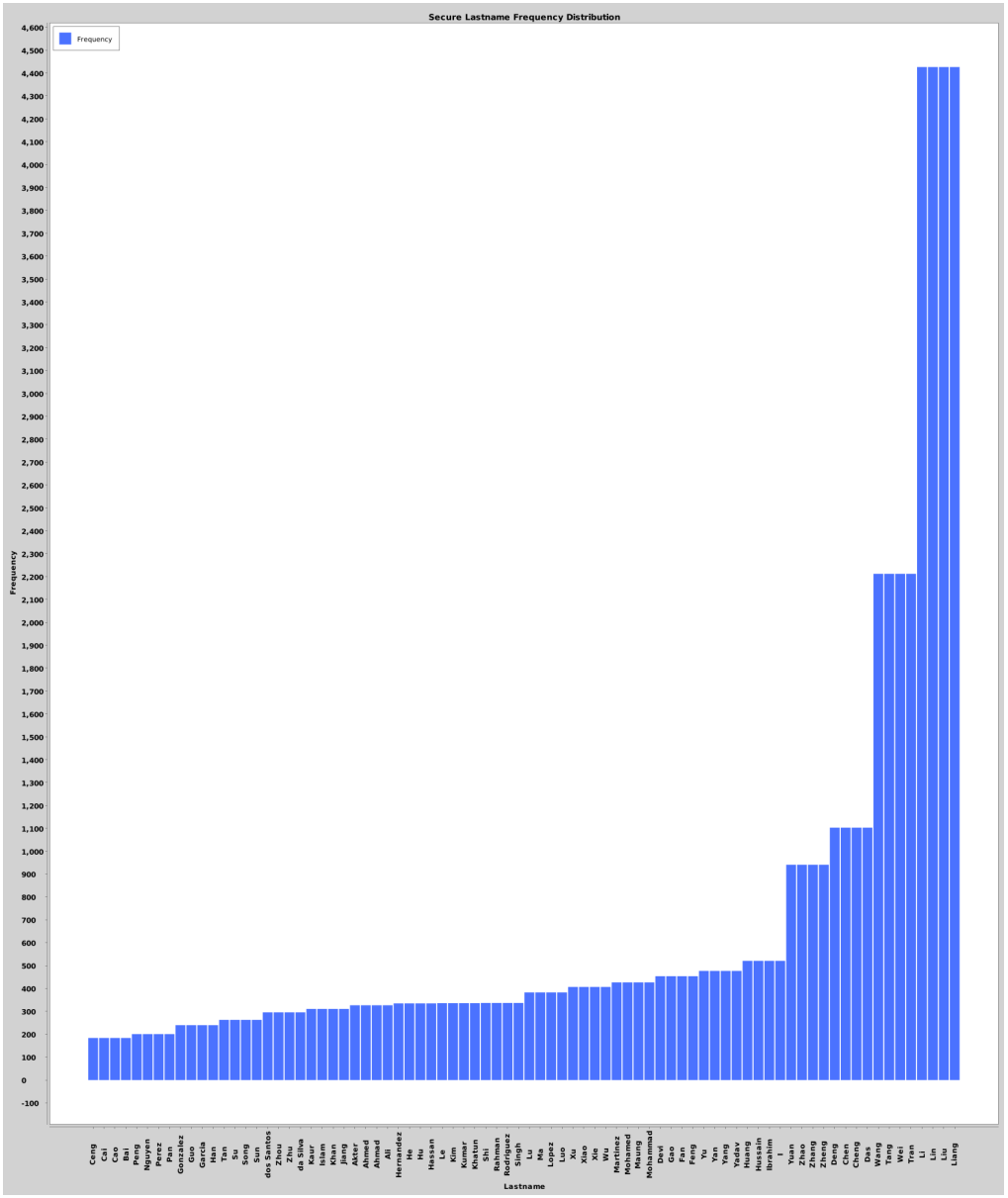

Figure 4: Distribution of encrypted last names.

```
97
98      public List<Student> getSecStudentsFromSurname(String surname,
            int alpha) {
99          List<Student> students = new java.util.ArrayList<>();
100         int i = surnameList.indexOf(surname);
101
102         //compute cluster boundaries
103         int clusterUpperBound = i;
104         for (int j = i; (j % alpha != 0 || j==i) && (j<rows); j++)
                {
105             clusterUpperBound = j;
106         }
```

```
107             int clusterLowerBound = i;
108             for (int j = i; (j % alpha != 0) && j>0 ; j--) {
109                 clusterLowerBound = j-1;
110             }
111 
112             //return students in alpha-size cluster
113             if(i != -1) {
114                 for (int j = i; j <= clusterUpperBound && (j<rows); j
                        ++) {
115                     for (int k = 0; k < cols; k++) {
116                         if (matrix[j][k]) {
117                             students.add(studentList.get(k));
118                         }
119                     }
120                 }
121                 i--;
122                 for (int j = i; j >= clusterLowerBound; j--) {
123                     for (int k = 0; k < cols; k++) {
124                         if (matrix[j][k]) {
125                             students.add(studentList.get(k));
126                         }
127                     }
128                 }
129             }
130 
131             return students;
132         }
```