Cryptanalysis of a class of ciphers based on Chi distributions

By: Dominique Elizee, Joel Harawa, and Eli Edme

This project was done by Dominique Elizee, Joel Harawa and Eli Edme. Elizee assisted with writing the report, writing the encryption code, and handling the input. Harawa assisted with writing the report, writing the encryption code, testing different distributions (ex. Chi distribution) to see if the values would be of use, and researched decryption strategies. Edme assisted with writing the decryption code and testing/researching decryption strategies, and We are submitting ... different cryptanalysis approaches. We made the following modifications, ...

For test 1, we used the texts in the 5 `plaintexts` vector and the ciphertext to do a chi square test comparing the values we would expect to see in a regular english text to the cipher text and seeing which one deviates the least.

All occurrences of characters in the plaintexts and the ciphertext in the vector of maps named allOccurrences. The maps for each candidate are filled with all the characters in the set SET. The initial counts are initialized to 0.

```
for (int i=0; i < plaintexts.length(); i++) {
        map<char, int> occurrences;
        for (int j=0; j < set.length(); j++) {
            occurrences[set[j]] = 0;
        }
        allOccurrences.push_back(occurrences);
}
```

The characters are then counted up for each candidate text and updated in each of the maps.

```
for (int i = 0; i < plaintexts.size(); i++) {
        for (int j = 0; j < plaintexts[i].size(); j++) {
            if (allOccurrences[i].count(plaintexts[i][j]) > 0) {
                allOccurrences[i][plaintexts[i][j]]++;
            } else {
                allOccurrences[i][plaintexts[i][j]] = 1;
            }
        }
```

```
 }
```

We have a `distributions` vector in which we store values for the chi square test. The values are calculated as follows.

```
vector<double> distributions;
    //calculates the chi squared values
    for (int i=0; i < allOccurrences.size()-1; i++) {
        double x_i = 0;
        int j = 0;
        for (auto it_o = allOccurrences[i].begin(), end_o =
allOccurrences[i].end();
             it_o != end_o;)
        {
            double e = plaintexts[5].size() *
EXPECTED_FREQUENCIES[j];
            double o_e = (it_o->second-e);
            double o_e2 = pow(o_e, 2);
            x_i += o_e2 /(double)e;
            it_o++;
            j++;
        }
        cout << "push back " << x_i << endl;
        distributions.push_back(x_i);
    }
```

The values in distribution and the plaintexts are then passed into a function named guess which finds the lowest chi squared value and corresponding plaintext, and outputs the best guess of the 5 plain texts

For test 2, we generated 100 different candidates, using 100 different keys, with different key lengths from 4 to 24. From those candidates we generated chi-square values by using the frequency distributions in a regular english text. The chi-square values were then compared with each other to see which was the most “english-like” candidate.

`test2Guess` has the ciphertext passed into it, and has a vector called `candidates` that is assigned to the vector returned by calling the `generateCandidates` function. `generateCandidates` has the ciphertext string passed into it, and an empty vector,

candidates, that is used to store the candidates for the guess. 100 keys are generated using two nested for loops.

```
candidates = [];
for (int i=minT; i < maxT; i++) {
     for (int j=0; j < candidatesPerKey; j++) {
          key = [];
          generate_key(i, key);
          candidate = encryptString(key, ciphertext);
          candidates.push_back(candidate);
     }
}
```

`encryptString` has the key vector passed into it. The keys were generated by calling the `generate_key` function, which produces a pseudorandom number between 0 and the max key length. `encryptString` uses the keys to shift characters in by the values in the key. The string vector returned from `generateCandidates` is assigned to string vector candidates in `test2Guess`.

`test2Guess` has a vector of doubles named distributions used to store the values from a chi squared test using the number of occurrences of each character in the candidates and the expected values in an english text.

A vector of maps (each map has characters and the amount of times they appear in a candidate text) allOccurrences is assigned to the return value of a function named fillOccurrences. The maps for each candidate are filled with all the characters in the set SET. The initial counts are initialized to 0.

```
vector<map<char, int> allOccurrences;
for (int i=0; i < candidates.length(); i++) {
     map<char, int> occurrences;
     for (int j=0; j < SET.length(); j++) {
          Occurrences[set[j]] = 0;
     }
}
```

The characters are then counted up for each candidate text and updated in each of the maps.

```
for (int i=0; i < candidates.length(); i++) {
     for (int j=0; j < candidates[i].length(); j++) {
          //check if character exists in candidate text
          if (allOccurrences[i].count(candidates[i][j]) {
               //increment if character already exists
               allOccurrences[i][candidates[i][j]]++;
          }
}
```

The values of the counts for each candidate are then used to calculate the chi square values for each candidate text. They are then stored in the vector of doubles named distributions.

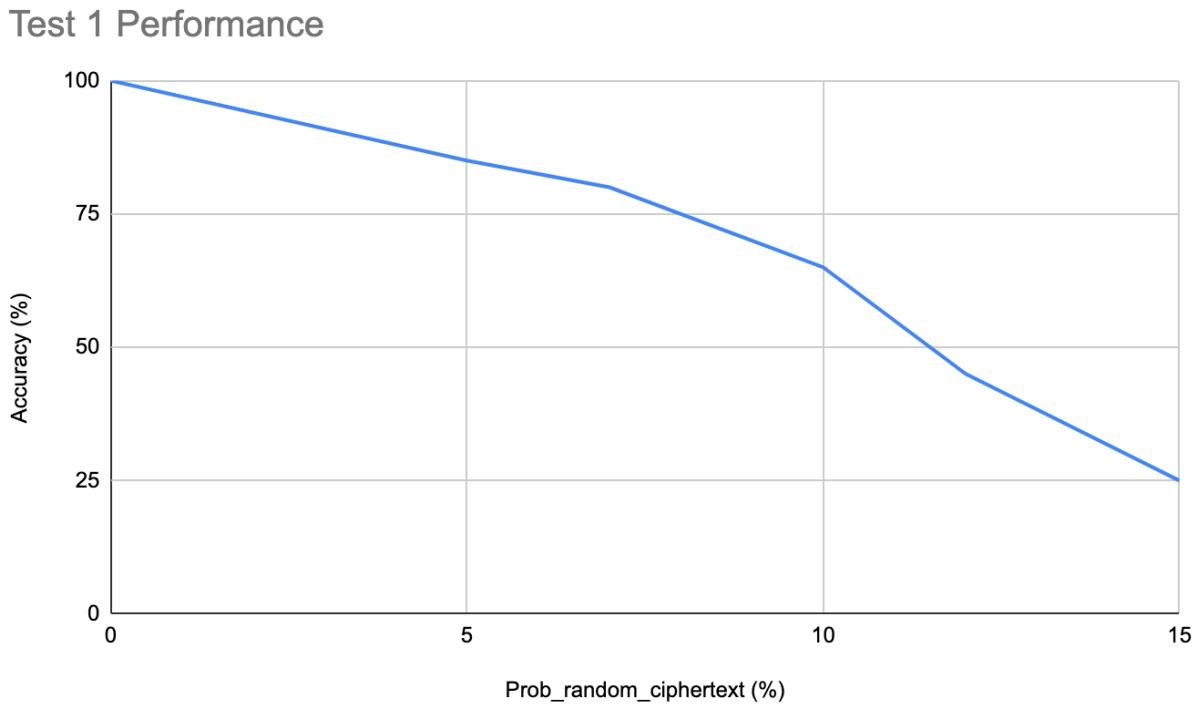
```
for (int=0; i < allOccurrences.size(); i++) {
     x_i = 0;
     k = 0;
     for (int j=0; j < allOccurrences[i].length(); j++) {
          exp = ciphertext.length() * EXPECTED_FREQUENCIES[k];
          obs_exp = allOccurrences[i]->second - exp;
          obs_exp2 = obs_exp * obs_exp;
          X_i += obx_exp2 / exp;
          k++;
     }
     distributions.push_back(x_i);
}
```

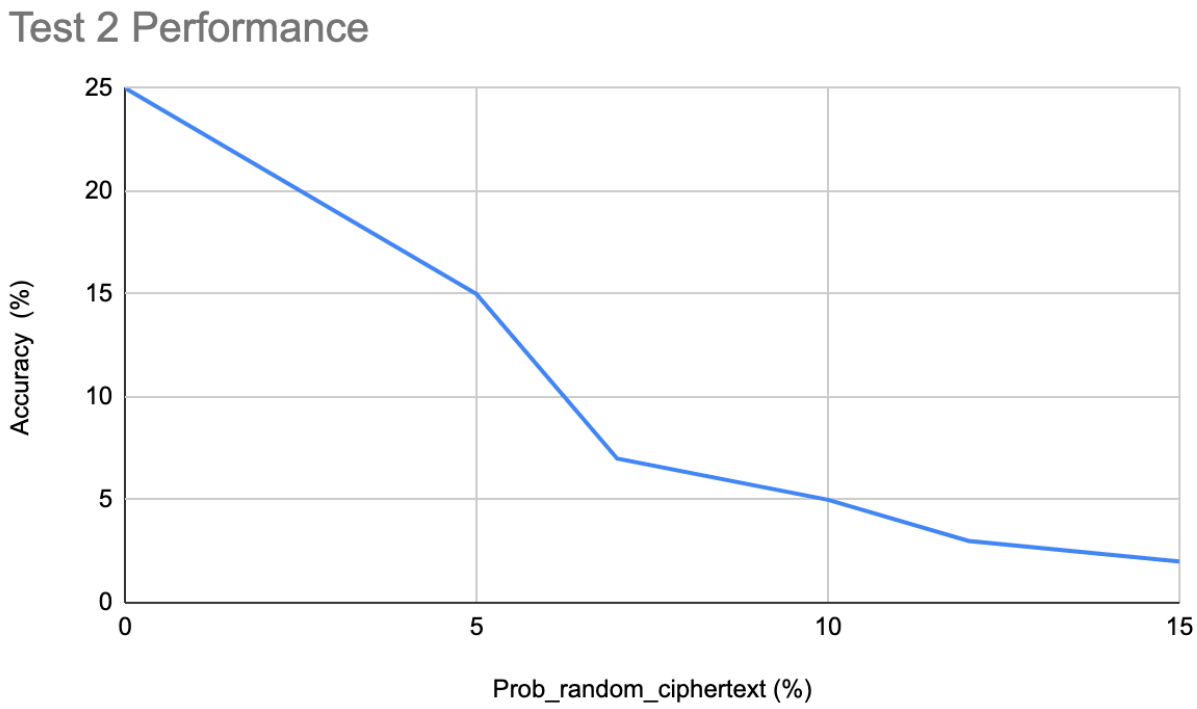
The second implementation of test two is based on the first one however instead of using encryptString to generate candidates, we shuffle the words in test to get as many permutations of what could possibly be the plaintext. The shuffling to get possible candidates is done in generateCandidates.

```
candidates = [];
totalCandidates = 1000;
totalWords = 50;
for (int j=0; j < totalCandidates; j++) {
     candidate = "";
     for (int i=0; i < totalWords; i++) {
          candidate += DICTIONARY2[rand % DICTIONARY2.length()];
          Candidate += " ";
```

```
        }
        candidates.push_back(candidate);
}
```

After developing the code and these strategies, we wanted to test their efficacy on randomly generated messages. The following visuals show their results.




Test 2 Performance
25
20
15
10
5
0
Accuracy (%)
0
5
10
15
Prob_random_ciphertext (%)