Crytanalysis of a class of ciphers based on character frequency comparison of possible plaintexts using Kasiski Examination

Introduction

This project was completed by Evan Richter, Matthew Mittelsteadt, and Alex Preneta.

Task	Description	Group Member
Create example schedulers	Created several example schedulers to be able to test with different variations of key scheduling algorithms	All
Key Length Guesser	Given a ciphertext, generate guesses at possible key lengths to be used in cracking the cipher	Evan
Cracking cipher with key length guesses	Given a key length and ciphertext, crack the ciphertext and output the best possible guess based on character frequency comparisons to the known dictionary	Alex/Evan
Spell check the possible plaintexts	Given a guessed plaintext and a dictionary, spell check the guessed plaintext to most closely match actual words in the dictionary.	Matt
Test algorithm	Use the scheduling algorithms to create tests for our code and look for opportunities to improve in order to yield better results.	All
Write Report	Write the report to outline techniques used and our approach	All

We are submitting a single cryptanalysis approach without modifications made to the above specifications. This cryptanalysis approach is a variation of the Kasiski Examination where the ciphertext is broken into n slices where n is the guessed length of the key.

Explanation of Approach

Our approach follows three basic steps: guessing the key length, forming an estimated decryption of the cypher based on the key length, and spellchecking that estimate to ensure its contents match the words in the given dictionary.

In our first step, the key length is guessed by dividing the ciphertext into equal length chunks and calculating the hamming distance between chunks. The hamming distance is calculated by counting the differences between the bits in each chunk. Using this hamming distance, we are able to generate an effective key length (length of the key after a scheduler is ran on it) by chunking the characters of the cypher text into various test sizes (possible key lengths). We then calculate the hamming distance between each chunk. Whichever chunk length (that is, possible key length) minimizes this score is likely the key length. This is because a randomly selected byte, on average, will have a higher score than an english letter byte as english letters exsist between the values 97 to 122 while a random byte exsists on the whole range of possible byte values, 0-256. If a test chunk size matches thev real key length, then the bytes it operates on will correspond to english letter values and therefore will yeild a low score.

Once the lowest scoring value is found, we presume it to be the correct key size and use it to guide subsequent steps.

Once a guess for the keylength is generated, we are able to attempt to crack the ciphertext by first dividing the ciphertext into `keylength` slices by taking every `i` th bit of the keylength and creating a slice with the corrosponding ciphertext bit. Once the ciphertext is divided into slices, we are able to crack each slice individually as if it was a single-byte key and calculate the character frequency of the slice. This character frequency can then be compared to a character frequency histogram generated on the given plaintext dictionary. The closest match beteen a size and a dictionary value will be returned in the output plaintext. Once every slice is cracked, the slices can be unsliced into a plaintext using the best character frequency match for each slice. The resulting plaintext should resemble plaintext words from the original dictionary, wlbeit with some errors.

The third step of our approach is to spell-check the guessed plaintext. This is done by dividing the guessed plaintext up into slices of possible words. For each slice, the levenshtein distance, which measure the total number of steps needed to compute one bytesteam into anthoer, is calculated for each plaintext word. THe matching value is returned and replaces the value in the origional guessed plaintext, presumably correcting any spelling errors. At the conclusion of this step, the best possible guess for the correct plaintext based on the most closely matching levenshtein distance of dictionary words is returned to give the final plaintext guess.

This process can be repeated to calculate multiple guesses in order to provide more opportunties to find the correct ciphertext; however, we found while testing that the top guesses closely matched the original plaintext before encryption.

Description of Approach

Derving Potential Key Lengths

The first step in our approach is to derive potential key lengths from the ciphertext. This is done by dividing the ciphertext into chunks and calculating the hamming distance between neighboring chunks:

```
chunks: Vec<chunks> = ciphertext.divided_into_chunks(chunk size);
let chunk1 = the first chunk
let chunk2 = the second chunk
for chunk1 in chunks:
  for chunk2 in chunks:
    distance += sum(chunk1 ^ chunk2)
```

This process is repeated for all possible key sizes between 3 and 120, values chosen as a practical guess. The hamming distance result as well as the keysize guess are pushed to a vector to compare later. Once this process is completed for all possible sizes, the keysize guesses are ranked by normalizing the hamming distances by dividing the hamming distance by the length of the smaller bit length text. We then sort results from shortest to biggest to find the lowest score.

Once these results are scored, the top result (that is, the lowest score value) is assumed to be the effective key length.

Cracking the cyphertext through letter frequency analysis

Using this assumed keylength, our cracking algorithm is run. For a given ciphertext-keylength pair, the ciphertext is divided into keylength sized slices:

```
for ct_index, ct_char in ciphertext:
  let bucket = ct_index % keylength
  ct_blocks[bucket].push(ct_char)
```

After slicing the blocks, we treat each block as a monoalphabetic substitution cypher and crack it using letter frequency analysis:

```
for letter in alphabet:
  for block in ct_blocks:
    plaintext = shift block by current letter
    calculate character frequency
    confidence = compare frequency to plaintext dictionary histogram
    CrackResult = {plaintext, confidence)
    return best confidence plaintext for block
  return vector of best confidence plaintexts for each block.
```

The slices are then assembled to form a single plaintext string:

```
for i in range(0, pt_blocks):
  for block in pt_blocks:
    pull out next character in the block
  return the unsliced plaintext
```

This process is repeated several times to form several guesses. The cracking function returns the plaintext with the best scoring confidence.

Spellchecking the guess to correct errors

Finally, the plaintext is spellchecked against the plaintext dictionary to remove words that remain jumbled in order to more accurately present them as actual plaintext. To accomplish this, we use an implementation of the levenshtein edit distance:

```
slice plaintext guess into probable words
for slice in plaintext_slices:
  for word in dictionary:
    distance = calculate levenshtein edit distance
    words = push(word, distance)
  best_word = min_distance(words)
full_plaintext_guess = push(best_word)
return full_plaintext_guess
```

The resulting plaintext is our final guess of the original plaintext which we output to the user.