Cryptanalysis of a Class of Ciphers Based on Key Probing and Statistical Analysis of Letter Distributions

By Leon Chou, Robert Kim

Introduction

In this write-up, we've developed 1 cryptanalysis approach, and are referring to that in this report. We used the ideas of probing to exit as early as possible, as well as taking a statistical approach with letter distributions to analyze potential keys for decrypting the ciphertext. We've also made no modifications to the project specification for the cryptosystem.

For our explanation and code in the following, we refer to "plainbytes" and "cipherbytes" as the plaintext and ciphertext encoded as an array of bytes, respectively, where each byte is the byte value of each character in the texts. We do the same for the key (and any other string in general), but still refer to it as the "key" for simplicity. This was done to make the math and interfaces in our program easier to handle by having a common format for our data.

Informal Explanation

Part 1

For part 1, since all 5 plainbytes are known, we take a brute force probing approach and try to exit as early as possible. We simply take all bytes *t* distance apart in the ciphertext, try to "decrypt" with each shift in the key space, and compare each decrypted set of bytes to all bytes *t* distance apart in each of the plainbytes. If we find an initial match, we continue by calculating a test key using the shift between the first *t* bytes in the plainbytes versus the cipherbytes. Knowing that the first *t* bytes of the cipherbytes is encrypted with the full *t*-length key was crucial for us to be able to use this method.

From our testing, this gave us a 100% success rate in decrypting the cipherbytes in a very efficient manner. If we could not detect that the cipherbytes could decrypt to 1 of the 5 potential plainbytes, we can safely assume the cipherbytes were encrypted from a message in part 2.

Part 2

For part 2, we took a statistical analysis across the cipherbytes, generating cases for each potential key-length. This approach allows us to isolate a single byte of the key and attempt to do statistical analysis of the possible searchspace. We compared the percentage of appearances of each character within the dictionary with the percentage of appearances of characters within the ciphertext. This allowed us to attempt to pick a single "shift" from the possible shifts by using the shift to decrypt, and then comparing the percentages of character appearances with our dictionary.

This allowed us to generate a "preferred" key for each possible key length, and we then decrypted using that key and attempted to verify our decrypted plaintext against the wordlist, to make sure that the decrypted plaintext was actually possible. If it was an impossible plaintext, we would attempt to retroactively fix our key a few times, picking a "fix" by selecting the character we had the next most confidence in.

If verification failed on every single possible key, we then took a confidence metric across all the "best" keys for each key length, used the length of the key as a heuristic so we could favor keys that had more chances of succeeding, and used that for our partial decryption.

Rigorous Explanation

Part 1

Our approach for part 1 was fairly straightforward, we used a probe across the given cipherbytes to exhaust a single byte of each key length, and if that byte had a single chance to decrypt, we then exhausted the search space for that specific key length. This allowed us to guarantee a decryption of part 1. Our entire implementation was done in 2 major functions, starting with `part1`.

```
def part1(cipherbytes, plaintexts):
 plainbytes = list(map(string_to_bytes, plaintexts))
 for t in range(1, KEY_MAX_LEN):
   plainseq = [(get_all_bytes_t_apart(p, t, 0), i) for i, p in
enumerate(plainbytes)]
   for k in range(len(KEY_SPACE)):
     # test current k using (k, t) across cipherbytes
     encrypted = get_all_bytes_t_apart(cipherbytes, t, 0)
     decrypted = decrypt(encrypted, [k])
     for seq, i in plainseq:
       if seq == decrypted: # does deep comparison
         part1decrypt(plainbytes[i], cipherbytes, t) # attempt decrypt
  return False
```

As can be noticed by our code, our high level algorithm only iterates over all potential key lengths, the size of the key space, and the number of plaintexts. Ignoring subroutines (which are at most linear in time), this is only at most 24 * 27 * 5 = 3,240 iterations, which is very fast given modern day computers.

```
def part1decrypt(plainbytes, cipherbytes, t):
 test_key = [offset(c, p) for p, c in zip(plainbytes, cipherbytes)][:t]
 result = decrypt(cipherbytes, test_key)

 if result == plainbytes:
   print(bytes_to_string(plainbytes))
   exit(0)
```

We attempt to actually decrypt the cipherbytes in the `part1decrypt` function, given our guess for t and for each set of plainbytes. Using our observation that the first t bytes are encrypted using the full t-length key, we simply find the offset between the first t bytes in the cipherbytes versus the plainbytes as our test key. If this succeeds, we are done. Otherwise, we continue probing other combinations.

<u>**Part 2**</u>

If part 1 fails to decrypt, we can safely assume that the cipherbytes were encrypted using the part 2 dictionary. For part 2, we took a statistical analysis across the cipherbytes.

```
def guess(cipherbytes, words):
 expected_distribution = gen_expected_dist(cipherbytes, words)
 pot_keys = guess_key(cipherbytes, expected_distribution)
 for key, _, top_fits in pot_keys:
   for _ in range(10):

     result = decrypt(cipherbytes, key)
     if verify(result, words):
       print('Success', key, len(key))
       return result

     key, top_fits = fix_key(key, top_fits)

 key = pot_keys[0][0]
 result = decrypt(cipherbytes, key)
 return result
```

We first generate an expected distribution from the cipherbytes and our part 2 dictionary.

```
def gen_expected_dist(cipherbytes, words):
 average_length_of_words = sum(map(len, words)) / len(words)
 expected_number_of_spaces = len(cipherbytes) // (average_length_of_words + 1)
 searchspace = ''.join(words)
 searchspace_bytes = string_to_bytes(searchspace)
 expected_distribution = calculate_letter_distributions(searchspace_bytes +
[ord(' ') for _ in range(int(expected_number_of_spaces))])

 return expected_distribution
```

We fetch the average length of words in the dictionary and use it to get a predicted number of spaces. Then we use it in `calculate_letter_distributions` along with all the words concatenated. This gives us an idea of the letter distributions of our dictionary, as well as how many spaces we should expect to see in the plaintext.

We then call `guess_key` using the calculated distribution. This in turn will iterate over potential key lengths, and attempt to generate a key with `gen_key`.

```
def gen_key(keylength, cipherbytes, expected_distribution):
 key = [0 for _ in range(keylength)]

 top_fits = []

 total_diff = 0
 for i in range(keylength):
   fits = best_fit(keylength, i, cipherbytes, expected_distribution)
   top_fit = list(filter(lambda x: x[1] > THRESHOLD, fits))
   best = fits[0]
   key[i] = best[0]
   total_diff += best[1]

   top_fits.append(top_fit)

 return key, total_diff / keylength, top_fits
```

We calculate the best key fits to the expected distribution byte by byte, and keep track of other potential values if their match is above `THRESHOLD`, or 0.5. We also keep track of our best guesses in `top_fits` so that we can use them later if our best key match is not successful. This is so we can get at least some partial match if possible. We also rank full keys based on total_diff, or the sum of how far apart key byte is from the expected distribution.

```
# generates a diff for each individual byte of a key
def best_fit(keylength, offset, cipherbytes, expected_distribution):
 _bytes = get_all_bytes_t_apart(cipherbytes, keylength, offset)
 distributions = []
 for key in KEY_SPACE:
   decrypted = decrypt(_bytes, [key])
   dist = calculate_letter_distributions(decrypted)

   diff = 0
   for k, v in dist.items():
     diff += abs(expected_distribution[k] - v)

   distributions.append((key, diff))

 return sorted(distributions, key=lambda a: a[1])
```

If our best guess succeeds at this point, we have found our plainbytes and are done. If our best guess fails, we call fix_key and replace each byte in the key with the next best fit until we can get a better decryption.

```
def fix_key(key, top_fits):
 best_fits = [(i, *fits[0]) for i, fits in enumerate(top_fits)]

 best_fits.sort(key=lambda x: x[2])

 i, k_byte, _ = best_fits[0]

 top_fits[i].pop(0)

 key[i] = k_byte

 return key, top_fits
```

If all of these fail, we can be certain we have failed to decrypt using our approach.
Then to attempt to give a semblance of a result back, we pick the key with the most confidence, and return the ciphertext decrypted using that key to the user.